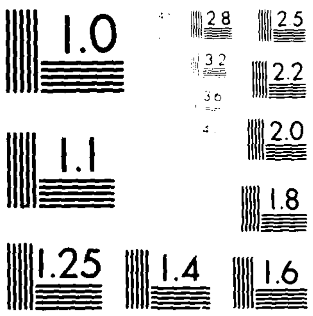
AD-A097 273 MARYLAND UNIV COLLEGE PARK DEPT OF COMPUTER SCIENCE F/G 9/2
THE OPERATIONAL APPROACH TO REQUIREMENTS SPECIFICATION FOR EMBE--ETC(U)
DEC 80 P ZAVE AFOSR-77-3181
UNCLASSIFIED TR-976 AFOSR-TR-81-0322 NL

1 of 1
AD A
097273



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

(1) LEVEL II

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER (19) (18) AFOSR-TR- 81-0322	2. GOVT ACCESSION NO. AD-A097273	3. RECIPIENT'S CATALOG NUMBER
4. ~~TITLE (and Subtitle)~~ (6) THE OPERATIONAL APPROACH TO REQUIREMENTS SPECIFICATION FOR EMBEDDED SYSTEMS.		5. TYPE OF REPORT & PERIOD COVERED INTERIM
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) (10) Pamela Zave (11) Dec 80 (15)		8. CONTRACT OR GRANT NUMBER(s) AFOSR-77-3181 ✓
9. PERFORMING ORGANIZATION NAME ~~AND ADDRESS~~ Department of Computer Sciences University of Maryland College Park MD 20742 (14) TR-976		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS (16) (17) 2304/A2 61102F
11. CONTROLLING OFFICE NAME AND ADDRESS Air Force Office of Scientific Research/NM Bolling AFB DC 20332		12. REPORT DATE DECEMBER 1980
		13. NUMBER OF PAGES 72 (12) 75
14. MONITORING AGENCY NAME & ~~ADDRESS~~(if different from Controlling Office) (9) Interim technical rept.,		15. SECURITY CLASS. (of ~~this report~~) UNCLASSIFIED
		15a. DECLASSIFICATION DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

DTIC ELECTE APR 2 1981 S D B

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Requirements analysis and specification, embedded (real-time) systems, system design and specification, simulation models, distributed processing, applicative programming.

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This paper is a self-contained and comprehensive presentation of the "operational" approach to requirements specification for embedded systems, which is based on the concept of specifying requirements as an executable model of the proposed system interacting with its environment. It is argued that urgent performance requirements characterize embedded systems, and a formal treatment of performance is given. The language is process-oriented so that the parallelism inherent in embedded system requirements can be expressed directly. Motivation and examples are emphasized throughout.

DD FORM 1 JAN 73 1473

UNCLASSIFIED 409022
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

AFOSR-TR- 81-0322

TR-976 December 1980

THE OPERATIONAL APPROACH TO REQUIREMENTS SPECIFICATION FOR EMBEDDED SYSTEMS*

Pamela Zave

Abstract This paper is a self-contained and comprehensive presentation of the "operational" approach to requirements specification for embedded systems, which is based on the concept of specifying requirements as an executable model of the proposed system interacting with its environment. It is argued that urgent performance requirements characterize embedded systems, and a formal treatment of performance is given. The language is process-oriented so that the parallelism inherent in embedded system requirements can be expressed directly. Motivation and examples are emphasized throughout.

Keywords requirements analysis and specification, embedded (real-time) systems, system design and specification, simulation models, distributed processing, applicative programming

---

*This research was sponsored in part by the Air Force Office of Scientific Research (AFOSR-77-3181■). Computer time was provided by the Computer Science Center, University of Maryland.

81 4 2 037

TABLE OF CONTENTS

0. INTRODUCTION 1

1. THE REQUIREMENTS PROBLEM 2

2. EMBEDDED SYSTEMS 6

3. AN "OPERATIONAL" APPROACH 11

4. QUALMS ABOUT OPERATIONAL REQUIREMENTS 23

5. THE PAISLEY LANGUAGE 31

6. INTERIM EVALUATION 49

7. PLANS FOR FUTURE RESEARCH 51

8. CONCLUSION 53

ACKNOWLEDGMENTS 54

REFERENCES 54

APPENDIX: A GRAMMAR FOR PAISLEY 58

FOOTNOTES 62

FIGURES 63

THE OPERATIONAL APPROACH TO REQUIREMENTS SPECIFICATION FOR EMBEDDED SYSTEMS

0. INTRODUCTION

Recently the study of system requirements has emerged as a major area of research in software engineering. It has become clear that the stated requirements for a system have tremendous impact on the quality and usefulness of the ultimate product, and on the efficiency and manageability of its development. Yet, despite their leverage, relatively little is known about deriving and specifying good sets of requirements.

At the same time, the prominence of "embedded" (roughly equivalent to "real-time") systems has been increasing, due largely to hardware advances which have made them feasible for a broader category of applications than ever before. We will argue that embedded systems are characterized by the urgency of their performance requirements; to the extent that all computer systems would benefit from the ability to state and satisfy performance requirements, even very specialized knowledge about embedded systems can be useful to developers of other types of system.

This paper presents a new approach to the problem of specifying the requirements for embedded systems. It offers a substantial increase in formality, expressive power, and usefulness (in terms of the kinds of processing that can be performed on, and the information that can be derived from, a requirements specification) over the current widely known requirements technologies.

This paper is self-contained. We give brief introductions to the subjects of requirements and embedded systems, explain and motivate our approach in detail, and then present a formal requirements language embodying it. Four systems, comprising a representative sample of embedded system structures and properties, are used in examples throughout. Finally there is an interim evaluation of the approach, and plans for future research.

1. THE REQUIREMENTS PROBLEM

1.1. The role of requirements in the system life cycle

The development of a computer system begins with the perception of a need for it. During the requirements phase, analysts should arrive at a deep understanding of that need, and propose a system to fill it. The product of the requirements phase is the requirements specification, which plays a unique and crucial role in the rest of development. It states what system is to be developed, at what costs, and under what constraints.

The project cannot be a complete success unless the requirements have the informed consent of everyone who will be involved, including members of the development organization (designers, programmers, and managers), the originating organization (the people who determine the cost and value of the system: managers of an operation--if the system is being developed for internal use, or salespeople--if the system will be offered as a product), and the ultimate users of the system. This consensus can only be achieved through feedback and negotiation, with preliminary versions of the requirements specification being the major vehicle of communication.

During design and implementation, the requirements specification defines the "top" for top-down design, and the product toward which management effort is aimed. At the end of development, it is the standard against which the system is compared for success or failure, acceptance or rejection.

Requirements are often neglected, for reasons that are all too familiar: lack of awareness of their importance (which is disappearing), lack of useful requirements analysis and specification techniques, and natural reluctance to incur costs and delays at the beginning of a project. Yet the consequences of this shortsightedness, which include cancelled projects or unprofitable products, unhappy users, chaotically structured systems, budget and schedule overruns as endless changes are made, and even lawsuits, are so serious that no one involved in software engineering can afford to ignore them. Other introductions to the role of requirements in system development can be found in [Boehm 76], [Bell & Thayer 76], [Ross & Schoman 77], [Yeh _et al._ 80], [Heninger 79], [Balzer & Goldman 79], and [Davis & Rauscher 79].

It should be noted that even with the most optimistic view of current progress on requirements analysis and specification, in which problems of communication and complexity can be solved, certain other problems will remain very difficult to deal with. One is that vital decisions must be based on forecasts of costs and even feasibility, while such forecasting is perhaps the weakest point of our software technology. Another is that the requirements are constantly changing, even as we try to write them down. And systems that are used evolve continually throughout their lifetimes ([Belady & Lehman 79]), creating "maintenance" costs which may eventually dwarf those of initial development.

As consciousness of the economic and technical importance of evolution in the system life cycle grows, we may develop a new concept of the life cycle based on iterated (re)developments, large and small, as in [Conn 80]. In such a model, the requirements specification will evolve with the system, serving throughout its life as definition, documentation, and contract. Needless to say, this expanded role will place even greater demands on the quality and modifiability of our requirements specifications.

1.2. Goals for requirements specifications

Progress in software engineering has almost always been made from the bottom up: from machine language to axiomatic specifications, for example, we have proceeded first by learning to do something, and then by understanding it well enough to find suitable abstractions of it. This paper takes the same approach to requirements. It seems unlikely that we will find really effective techniques for requirements analysis before we know how to write a good requirements specification recording the results of that analysis. Therefore we will concentrate on specification techniques (although useful results on specification cannot help but suggest analytic methods and principles).

The characteristics of a good requirements specification can be inferred from the things that will be done with it. Since the latest version (to call it a "final" version is to ignore the reality of system evolution) must be obtained by iterative communication and negotiation, the specification must be modifiable. It must also be easy for people to understand.

What would make a specification understandable? Perhaps the biggest barrier to understanding large systems is complexity, and so the specification must decompose complexity in every appropriate way. Three forms of decomposition are already familiar in various contexts: abstraction, partition, and projection (Figure 1). Abstraction forms a hierarchy of representations in which detail is suppressed at the higher levels and elaborated at the lower levels. Partition is used to represent the whole as the sum of its parts, making it possible to examine the parts one at a time. Projection represents the whole, but only with respect to a subset of its properties. The obvious example of a projection is a two-dimensional architectural drawing of one view of a three-dimensional building. A requirements specification language must support all three kinds of decomposition, alone and in combination.

Good decomposition of complexity (procedures, data modules, monitors, etc.) makes it possible to understand programs by reading them. The other way that people come to understand programs is by testing them. Testing is so essential to programming that it seems foolish to do without it at any stage of development. Therefore requirements specifications should be executable, and thus subject to validation by testing. The potential benefits are very great, because executable requirements can be "debugged", used to put on behavioral demonstrations for customers, turned into "fast prototypes", and more (see 3.3).

Finally, a specification intended to be understood by people should be intuitive, i.e. it should be written so as to assist the memory and elicit tacit knowledge (see the "human factors" section of [Yeh et al. 80] for a survey of psychological findings concerning requirements).

The other major purpose for which a requirements specification is used is constraining the target system of the development project. To do this well it should be precise, unambiguous, internally consistent, and sufficiently complete. It should also be minimal, i.e., define the smallest set of properties that will satisfy the users and originators. Otherwise the specification may over-constrain the target system, so that some of the best solutions to design problems are unnecessarily excluded.

The specification must also be used to accept or reject the final product. If verification is to be used for this purpose, the specification must be formally manipulable and therefore formal (although formality has already been implied by precision, lack of ambiguity, consistency, and executability). If acceptance testing is to be used, the testable behavior of executable requirements will provide a concrete standard to which the implementation can be compared.

The remainder of this paper is concerned with a requirements specification approach (and language) that promises to help us achieve many of these goals. It is also somewhat specialized for a particular class of systems, namely . . . .

2. EMBEDDED SYSTEMS

Common examples of embedded systems are industrial process-control systems, flight-guidance systems, switching systems, patient-monitoring systems, radar tracking systems, ballistic-missile-defense systems, and data-collection systems for experimental equipment. The class of embedded systems is an important one, partly because it already includes some of our oldest and most complex computer applications, and partly because it is expanding rapidly in volume and variety as a result of the microprocessor revolution.

2.1. What makes a system "embedded"?

The term "embedded" was coined by the U.S. Department of Defense in conjunction with its common language (Ada) development project. "Embedded" refers to the fact that these systems are embedded in larger systems whose primary purposes are not computation, but this is actually true of any useful computer system. A payroll program, for instance, is an essential part of a business organization, which is a system whose primary purpose is selling products at a profit.

The common concept that unites the systems we choose to call "embedded" is process control: providing continual feedback to an unintelligent environment. This "theme" is easily recognized in flight-guidance systems and switching systems; even in a patient-monitoring system, sick patients are not exercising their intelligence in interacting with the system, and nurses can be viewed as providing a mechanical extension to the system's feedback loop.

The continual demands of an unintelligent environment cause these systems to have relatively rigid and important performance requirements, such as real-time response requirements and "fail-safe" reliability requirements. It seems that this emphasis on performance requirements is what really characterizes embedded systems, and causes us to be more aware of their environments than we are for other types of system.*

Figure 2 shows an informal classification of systems, based on properties that show up at the requirements level. Requirements for "support systems" are generally much less definite than requirements for applications systems.

And while the performance requirements for embedded systems may be couched in absolutes, the performance requirements for support systems will be relative to resources and resource utilization, and the performance requirements for data-processing systems will be relative to load, resources, and psychological factors. The most complex systems, such as nationwide airline-reservation systems, should probably be viewed as having subsystems of all three types.

2.2. The special problems of embedded systems

The special nature of embedded systems exacerbates many software engineering problems, and thus demands particular attention even during the requirements phase.

Few organizations have logged as much experience with embedded systems as the Department of Defense, which spends 56 per cent of its approximately 3 billion dollar annual software budget on them ([Fisher 78]). Here is a pointed summary of that experience:

> Embedded computer software often exhibits characteristics that are strikingly different from those of other computer applications. The programs are frequently large (50,000 to 100,000 lines of code) and long-lived (10 to 15 years). Personnel turnover is rapid, typically two years. Outputs are not just data, but also control signals. Change is continuous because of evolving system requirements--annual revisions are often of the same magnitude as the original development ([Fisher 78]).

Clearly coping with complexity and change will not be easier in the domain of embedded systems.

In addition to the performance requirements, which have already been established as a major distinguishing factor, embedded systems are especially likely to have stringent resource requirements. These are requirements on the resources, mainly physical in this case, from which the system is constructed. This is because embedded systems are often installed in places (such as satellites) where their weight, volume, or power consumption must be limited, or where temperature, humidity, pressure, and other factors cannot be as carefully controlled as in the traditional machine room.

The interface between an embedded system and its environment tends to be complex, asynchronous, highly parallel, and distributed. This is another direct result of the "process control" concept, because the environment is likely to consist of a number of objects which interact with the system and each other in asynchronous parallel. Furthermore, it is probably the complexity of the environment that necessitates computer support in the first place (consider an air-traffic-control system)! This characteristic makes the requirements difficult to specify in a way that is both precise and comprehensible.

Finally, embedded systems can be extraordinarily hard to test. The complexity of the system/environment interface is one obstacle, and the fact that these programs often cannot be tested in their operational environments is another. It is not feasible to test flight-guidance software by flying with it, nor to test ballistic-missile-defense software under battle conditions.

2.3. Representative examples of embedded systems

The following systems, when developed appropriately, represent a wide variety of structures and problems typical of embedded systems. They will be used in examples throughout the paper.

2.3.1. An airline-reservation system

This is an on-line (interactive) database system used for airline reservations. It is accessed from 10,000 terminals across the country, and must process an average of 200 transactions per second (these and other quantities are taken from [Knight 72]).

Systems of this type are not always treated as embedded--rather, their data-processing nature is emphasized, and the users are expected to accomodate themselves to whatever level of performance the system offers. We emphasize its embedded-system characteristics by requiring certain absolute levels of performance, and by taking the physical effects of geographical dispersion into account at the requirements level.

2.3.2. A process-control system

This system monitors three machines in a factory; while fairly simple, it has an interesting variety of activities. A relatively complete specification of it can be found in [Zave & Yeh 81].

Conditions local to individual machines may call for minor adjustments, which are done automatically by the system. Conditions arising in the factory as a whole, however, may be quite dangerous, and are responded to by a human operator. The system's responsibility is to detect these conditions and sound an alarm.

The system also keeps information about factory conditions, which is used for two purposes: printing reports on production and consumption of raw materials, and answering queries about machine and factory status from the operator (particularly when the alarm has sounded).

2.3.3. A patient-monitoring system

Patient-monitoring systems are often used as examples in the requirements literature, although the usual treatment is naive. (Primarily, only the data-oriented aspects of the system are considered.) The system reads sensors attached to patients in an intensive-care unit of a hospital. The system displays a warning message on a CRT screen if a sensor value falls outside of a safe range or if a sensor appears to be malfunctioning. Interesting sensor values are stored in a database which can be queried from the terminal. The frequency with which a sensor is read, the safe range for a sensor value, and the criteria for keeping readings in the database, can all be adjusted by doctors and nurses from the terminal.

2.3.4. The Air Force Weapons Effectiveness Testing (AFWET) system

This system (ultimately realized under the name "WESTE") was an early real-time system which suported quantitative testing of U.S. military (conventional warfare) capability (see Figure 3). Its requirements document ([Air Force 65]) is a fruitful source of bad examples and unsolved requirements problems.

Tests were military exercises involving "test elements" such as airplanes, ships, tanks, and ground defense positions (some playing the role of enemy forces), confined to a circle centered on Eglin Air Force Base in Florida. Test elements communicated with a central site through military standard radio equipment, plus a contractor-supplied communications network.

During a test, moving elements would send periodic notifications of their positions to the central site. Mock firings of weapons would also cause messages to be sent, supplying all relevant parameters such as the direction of aim. The central system would simulate the battle in real time, determining which of the mock firings would have resulted in "kills". The results of the simulation were (a) used to display the course of the battle on graphics scopes for the benefit of officers in a control room, (b) dumped onto archival storage for later analysis, and (c) used to send "kill" notifications to "killed" test elements in the field. They would then react with a flashing light or loud noise, and cease to participate in the battle.

3. AN "OPERATIONAL" APPROACH

The approach taken in this paper is to specify the requirements for an embedded system with an explicit model of the proposed system interacting with an explicit model of the system's environment. Both submodels consist of sets of asynchronously interacting digital processes, although some of the processes in the environment model may represent discrete simulations of nondigital objects such as people or machines. The entire model is executable, and the internal computations of the processes are specified in an applicative language.

We call this the "operational" approach because the emphasis on constructing an operating model of the system functioning in its environment provides its primary flavor. It has been embodied in a Specification Language which, since it is based on the ideas above and is therefore Process-oriented, Applicative, and Interpretable (executable), is named PAISLey.

In the remainder of this section, the basic ideas behind the operational approach will be explained, illustrated, and justified in detail. Section 4 addresses the apparent disadvantages of the operational approach, and Section 5 defines PAISLey.

3.1. Explicit modeling of the environment

Figure 4 is a diagram of the processes and interactions in part of the requirements model for a patient-monitoring system. The "patient", "nurse", and "doctor" processes are all digital simulations of these natural objects, representing (obviously) only the roles played by these people with respect to patient-monitoring. Thus the nurse's behavior includes only (a) treating a patient because of a warning from the system, (b) adjusting a sensor because of a warning from the system, and (c) interacting with the system (via the "crt-terminal" process) in any way requested of him during interactions with a doctor.

The "sensor" and "crt-terminal" processes represent analog-to-digital conversion devices. They are considered parts of the environment rather than the proposed system simply because they are "given": the contractor need not supply them, nor can he change what they are.

The "<u>reader</u>" process reads (and timestamps) sensor data at intervals of real time specified from the terminal. The "<u>monitor</u>" process checks the data according to criteria specified from the terminal, sending warning messages if a health factor falls outside the safe range, or if a sensor seems to be malfunctioning. It also sends sufficiently interesting data to the "<u>database</u>" process, which responds to queries from the terminal and also purges old data to maintain itself at a reasonable size.

Including an explicit model of the environment has several advantages for requirements specification. The reason that the interface between an embedded system and its environment is complex, asynchronous, highly parallel, and distributed is that it consists of interactions among a number of objects which exist in parallel, at different places, and are not synchronized with one another. Organizing these interactions around the objects (processes) which take part in them is an effective way to decompose this sort of complexity. Furthermore, assumptions and expectations on both sides of the boundary can be documented. The result is a specification which is far more precise and yet comprehensible than could be obtained by treating either side of the interface as a "black box", which is what happens when the environment is not modeled.

Another reason for having an environment model is that the environment (when construed broadly enough) is the source of all changes to the system. Modeling it is therefore a promising way to anticipate changes and enhance the modifiability of both specification and target system.

A simple, but not unimportant, example of this has to do with the environment/system boundary. After requirements for a patient-monitoring system along the lines of Figure 4 have been specified, it may be decided that the contractor should supply terminals and sensors after all. The change to the specification itself will be trivial, since the boundary is arbitrarily placed, and not really part of the executable model. More importantly, most of the "new" analysis work will have already been done: the analysts will understand <u>fully</u> (i.e. from both sides) the function of this equipment, and will probably be very aware of any shortcomings that should be corrected, now that the freedom exists to do so.

The final advantage of specifying the environment is that many performance constraints are most naturally attached there. The patient-monitoring system has (among others) response-time requirements on database queries, and a requirement to be able to handle a certain load of sensors and terminals. The response requirement is most directly expressed as a time limit on the component of the terminal specification which waits for the response after sending a query. The load is largely a function of the numbers and output rates of sensors and terminals, and so specifications of sensors and terminals must be a large part of specifying any load requirement.

The other significant aspect of constructing an environment model is that it is a valuable tool for requirements analysis, as well as specification. In fact, the best way to analyze requirements may be to start with the environment model, and work "outside-in" to a proposed system which supports a desirable mode of operation in the environment. The extreme case is automation of an existing manual system--in the absence of changes to existing procedures, the requirements can be derived simply by modeling the current operation, and drawing a boundary to distinguish the automatable part! [Yeh et al. 79a] and [Yeh et al. 79b] both discuss "conceptual models", which are models of system environments constructed for the purpose of requirements analysis.

In the patient-monitoring system, since only the "sensor" and "crt-terminal" processes interact directly with the proposed computer system, only these are necessary for precise specification of the system interface. The "patient", "nurse", and "doctor" processes appear strictly as vehicles for requirements analysis. Wondering how doctors and nurses interact leads the analyst to ask which kinds of information a doctor expects to get from a nurse on duty, and which kinds he would like to find in the database. Wondering how nurses interact with patients and the display leads the analyst to ask how the display screen should be allocated to medical histories versus emergency messages, how often warnings concerning an ongoing crisis need be displayed, and whether information from the monitoring system is needed at the patient's bedside. These questions are never asked (or answered) in the numerous treatments of patient-monitoring systems appearing in the requirements literature.

Even if the analysts can achieve understanding of the requirements in some other way, early concentration on the environment may lead to better communication with users (who are much more interested in their environment than your system), and more open-minded problem-solving, unbiased by preconceived notions or similar systems the analysts have worked on.

3.2. Processes

Another key feature of the operational approach is that the primary units of specification are processes. A process is a simple, abstract representation of autonomous (distributed) digital computation. It is specified by supplying a "state space", or set of all possible states, and a "successor function"** on that state space which defines the successor state for each state. It goes through an infinite sequence of states (although a "halting" process can be specified by having it go into a distinguished "halted" state which it will never leave), asynchronously with respect to all other processes (Figure 5).

A process is cyclic, with its successor function describing its natural cycle. The natural cycle of a process simulating a sick patient, for instance, would be a single step of the discrete simulation algorithm. The successor function of such a process might be declared as:

patient-cycle: PATIENT-STATE --> PATIENT-STATE,

where the set "PATIENT-STATE", which is its domain and range and also the state space of the process, contains values encoding possible states of the patient between simulation steps. The natural cycle of the "doctor" process might be to take one action, either asking one question of a nurse, giving one order to a nurse, or taking part in one transaction with the patient-monitoring system.

There can be no question about the generality of processes. They were originally used as abstractions of concurrent activities within multiprogramming systems ([Horning & Randell 73]), and many recent articles have shown that they can be used to represent I/O devices, data modules, tasks, monitors, buffers, or any other identifiable structure within a computer system (e.g. [Hoare 78], [Brinch Hansen 78], [Mao & Yeh 80]). Process-based models of computation have been the focus of extensive theoretical work and the language Smalltalk ([Ingalls 78]). Our varied examples are persuasive evidence that the notion of digital simulation of nondigital objects is similarly powerful in describing the environments of computer systems.

The appropriateness of using processes to specify requirements for embedded systems is based on our observation that in these systems, asynchronous parallelism--among environment objects, between environment objects and the system, and within the system (if only for reasons of performance)--occurs naturally at the requirements level. One happy result of recognizing that parallelism, by using processes as the specification unit, is environment specifications which should be highly intuitive, even to naive users. This is because they are populated by identifiable models of the same autonomous, interacting objects from which the real world is made.

Perhaps the best way to appreciate processes is to consider the alternatives: representations of processing found in other requirements languages. The one most commonly found in requirements documents is "dataflow". Dataflow diagrams show major system functions, and identify the data structures which are their inputs and outputs (e.g. Figure 6). Dataflow is the basis of PSL/PSA ([Teichroew & Hershey 77]) and SADT ([Ross 77], [Ross & Schoman 77]), and has probably been rediscovered thousands of times by isolated requirements-writers.

Dataflow may be adequate for many data-processing systems, such as the one depicted in Figure 6. This is because major subfunctions ("check-inventory", "send-invoice") are implemented as major subprograms, and subprograms are invoked in some implicitly understood sequence, whenever their input files are ready.

Dataflow is seriously inadequate for embedded systems, however, because control is all-important in embedded systems, and takes a variety of forms which are not captured by the simplistic notion of control implicit in dataflow. If the system in Figure 6 had the "on-line" character and performance requirements of an embedded system, here are some of the problems we might encounter with the dataflow approach: (1) A distinction must be made between inputs which are always present (such as the "INVENTORY" database) and inputs which invoke a function whenever a new instance appears (such as "PURCHASE-ORDER"). The situation is even more complex when there is an input value (such as the current output of a sensor attached to a hospital patient) which is always available, but only read at certain real-time intervals (and the interval itself is a variable stored in some system database). (2) Functions (such as "process-account-order" and "process-payment") may have to be executed concurrently to meet performance requirements, in which case they must synchronize their uses of shared resources or databases (such as "ACCOUNTS"). (3) Functions may no longer execute in a predefined sequence (because of simultaneous access from multiple terminals, the need for internal housekeeping, etc.), and so a complex interplay of events and states must be anticipated. With so many departures from the kind of information directly expressable in a dataflow diagram, it becomes less and less likely that dataflow can provide a meaningful characterization of the system.

The control arrow in SADT adds an explicit representation of control to dataflow diagrams (an illuminating discussion of its significance can be found in [Ross 77]), but its informality prevents it from being precise or expressive enough for embedded systems. Processes and their interactions, on the other hand, are well-suited to the task of specifying complex control, as would be expected from their historical origins in the specification of operating systems.

Other notions of control appearing in requirements languages are stimulus-response paths in RSL ([Bell et al. 77], [Alford 77], [Davis & Vick 77]) and finite state machines ([Heninger 79], [Davis & Rauscher 79]). A finite state machine is very much like a single process--better than dataflow, perhaps, but permitting no explicit parallelism, decomposition of complexity, nor modeling of the environment.

Stimulus-response paths (e.g. Figure 7) do make it possible to decompose the requirements and represent parallelism. The "R-net" in Figure 7 shows explicit parallelism between "STORE_FACTOR_DATA" and "EXAMINE_FACTORS", and is only one of several R-nets specifying the entire system. They do not, however, provide for representation of data, the interaction of paths via data, or internal synchronization around shared data. Because the process mechanism integrates data and processing, it is a more complete formalism, and therefore more likely to be able to cope with a variety of systems and situations.

3.3. Executability

In the operational approach, requirements specifications are executable. This means that, under interpretation, the specification becomes a simulation model generating behaviors of the specified system.

It is of vital importance to be able to interpret specifications regardless of their level of abstraction. Not only are requirements by their nature abstract in many respects, but they must also be developed by successive refinements of understanding, each version of which should benefit from this facility. We will defer until 3.4 a discussion of how this can be done, and only deal here with the advantages of doing so.

Executable specifications can be tested. As mentioned in 1.2, this means that they can be debugged by the analysts who write them, and then validated by originators/users in demonstrations. Note that this capability includes a "fast prototyping" facility, which is now being mentioned by many authors as a valuable engineering tool ([Conn 80]), because the specification can be developed to whatever level of detail is appropriate for a prototype and then made available for use by a small community via the interpreter.

The ability to test is no panacea, as must be obvious from the literature on program testing--testing cannot demonstrate the absence of errors, it is not always easy to get the right kind of output from a test, and it is difficult to draw any general conclusions about a program on the basis of tests. And with embedded system specifications, there is the additional complication that any test must choose one of many relative-rate-dependent process execution sequences. Nevertheless, the problems inherent in testing programs have never caused us to give up testing them, and it seems plausible that requirements testing, once established in common practice, would seem likewise indispensable.

Furthermore, an executable requirements model can continue to be useful after the requirements phase. The environment part of the model can be used as a test bed during system development, which will be particularly valuable for embedded systems because of the aforementioned difficulties of testing them "in the field" (in fact, it is almost always necessary to write an environment simulator for exactly this purpose). The model of the proposed system can be used to generate sample behaviors for acceptance testing.

It is also possible to attach performance constraints in such a way that they can be simulated along with the functional requirements, and this is done in PAISLey. Simulation can then be used to predict performance where it is too complex to determine analytically. This type of simulation is an important feature of SREM, the integrated set of tools by which RSL is supported.

There is a final, critically important, advantage of executability that has nothing to do with testing or simulation. It is that the demands of executability impose a coherence and discipline--because the parts of a specification must "fit together" in a very strong sense--that could scarcely be obtained in any other way. If an executable requirements specification is shown to be internally consistent, that means it will continue to generate behaviors without ever halting, deadlocking, or going into an undefined state. In other words, it is guaranteed to be a valid specification of some system interacting with some environment. Clearly this is the utmost that any formally defined notion of internal consistency could do for us, since deciding whether they are the right system and environment is a matter of validation by the originator/user, or verification of consistency with externally defined axioms of correctness.

3.4. Specification in an applicative language

Within a process, computation (i.e. the successor function of the process) is specified using a purely applicative language. "Applicative" (or "functional") languages are those based on side-effect-free evaluation of expressions formed from constants, formal parameters, functions, and functional operators ("combining forms" for functions, such as composition). Well-known examples of applicative languages are the lambda calculus, pure LISP, and the functional programming systems of [Backus 78].

3.4.1. Advantages of applicative languages

Applicative languages are currently receiving a great deal of favorable attention because of their numerous theoretical and practical advantages ([Backus 78], [Iverson 80], [Smoliar 80], [Friedman & Wise 77], [Friedman & Wise 78a], [Friedman & Wise 78b], [Friedman & Wise 79], [Friedman & Wise 80], among others), most of which can be exploited in requirements specifications. To begin with, because applicative languages are interpretable, they support the executability property: processes are executed by repeatedly replacing their current states by successor states, and successor states are discovered by interpreting the applicative expressions which define them.

For purposes of high-level specification, the most important property of applicative languages is their tremendous powers of abstraction, i.e. of decision deferment. Consider, for instance, the functional expression "f[(g[y],h[z])]", which says that the function "g" is to be applied to the argument "y" and "h" is to be applied to "z" (the "[ ]" symbols denote function application or composition), and then "f" is to be applied to their values (the "( )" symbols are used to construct tuples of data). But it does not constrain the data, control, processor, or other resource structures used to do so. Are "g[y]" and "h[z]" evaluated sequentially or in parallel? In what data structures are their values stored? Perhaps the arguments "y" and "z" are even shipped off to special "g"- and "h"-processors, respectively, at different nodes of a network!

Furthermore, a primitive function has several interesting interpretations, all of which enable additional decompositions of complexity. A primitive function can represent a set of deferred decisions, to be made later by defining the function in terms of simpler primitives. It can also represent a mapping which will always remain nondeterministic from the perspective of the requirements model, because it depends on factors outside the scope of the model. For instance, in specifying a terminal we might declare a primitive function

think: DISPLAY --> INPUT,

where "DISPLAY" is the set of all CRT screen images and "INPUT" is the set of all input lines, to represent the human user's thought processes. Finally, in PAISLey a primitive function can be an abstraction for an asynchronous process interaction (see below). Because of these many options, applicative languages have been used successfully to describe phenomena ranging in level of abstraction from digital hardware to distributed system requirements ([Fitzwater & Zave 77], [Smoliar 79]).

An interpreter for an abstract specification language makes expedient and non-functionally-significant decisions about such matters as control and space allocation. The only other thing needed for interpretation is some sort of implementation of functions and sets left primitive in the abstract specification. This can be done in many ways, perhaps the simplest of which is: (1) any evaluation of a primitive function whose range is not primitive yields either a randomly chosen, or a "smallest", element of the range; (2) any primitive set is temporarily defined to be the set "FILLER", whose only element is the constant "´null´" (thus any evaluation of a primitive function whose range is primitive necessarily yields "´null´"). "´null´" is often used as a place-holder where a value must be generated but no semantics need be carried. Another way to interpret primitive functions is to display their arguments at a terminal and ask the analyst to supply a value, thereby creating an interactive testing system. In either case, the effect is to simulate the decisions which have been made, without interference from the decisions that haven't been made.

Another advantage of applicative languages is that they are extremely convenient for formal manipulations such as verification. This is because an expression has "referential transparency", i.e. its only semantic property is its value. An applicative program can sometimes be proven consistent with an axiomatic specification of correctness, for example, merely by algebraic substitution! This facility is one of the major subjects of [Backus 78].

One advantage of applicative languages that will not be exploited for specifications of embedded systems is that as programming languages, applicative languages may have more potential for efficient implementation than procedural ones. Because the "von Neumann bottleneck" of accessing and referring to memory one word at a time has been eluded ([Backus 78]), the field is clear for high-powered optimization by interpreter writers and machine designers. The work of Friedman and Wise on large-scale multiprocessing ([Friedman & Wise 78a]) and research on dataflow computers are both efforts in this direction; neither form of parallelism requires the knowledge or participation of the programmer.

Embedded systems do not profit directly because they may have performance, accessibility, distribution, etc. requirements which can only be simulated (but never realized) by an interpreter running on a conventional shared computer system, even though that interpreter might provide a convenient and efficient implementation for other types of system. For embedded systems, interpretable <u>specifications</u> are clearly distinguishable from <u>implementations</u> by their performance and resource properties, despite functional equivalence.

3.4.2. PAISLey as an applicative language

PAISLey is not a purely applicative language because states in general, and process states in particular, are not applicative concepts. System specifications can be written in a purely applicative notation, as in [Smoliar 79] and [Friedman & Wise 79]. In [Zave 80a] it is explained that, while many aspects of even embedded systems can be specified applicatively, the specification of most performance requirements, real-time interfaces with the environment, and certain resource requirements, all necessitate the introduction of some non-applicative structure such as processes. Furthermore, processes may offer a helpful form of decomposition of complexity not available in purely applicative languages when feedback loops are present.

This will be particularly important with our requirements specifications for embedded systems, in which representation of the feedback provided by the proposed system to the environment is a primary objective.

Since PAISLey is a blend of the applicative world and the non-applicative world of processes and states, the "seam" must be a smooth one. The two worlds meet at the mechanism for interprocess interaction, which is necessitated by the existence of processes, but designed to fit smoothly into the applicative framework. Interactions take place through a set of three primitives called "exchange functions" which carry out the side-effect of asynchronous interaction, but look and behave locally (intraprocess) exactly like primitive functions. Exchange functions are defined and explained in 5.3. They are a unique mechanism which seems to fulfill our purposes very well, and also offer an interesting new perspective on asynchronous interaction mechanisms for distributed processes.

Applicative languages have a reputation for unreadability, and general unsuitability for large-scale software engineering, in some circles. We believe that this reputation is due to typelessness and recursion, neither one of which is present in PAISLey.

Recursion is what purely applicative languages use to specify repetitive computation, and is analogous to looping (iteration) in procedural languages. Both are analogous to the repetitive application of a successor function to produce successive process states, which is how unbounded repetition is specified in PAISLey.

In most applicative languages, the only type of data object is the list or sequence, and all functions are applied to one list and produce one list. Since every function should be prepared to accept argument lists of any internal structure, there must be a distinguished "undefined" value produced whenever the internal structure of the argument is unsuited to the semantics of the function (as in [Backus 78])--and this mismatch must first be detected! Multiple arguments to or values from functions must be packaged in single lists, yet the existence of this substructure (or any other substructure, for that matter) cannot be explicitly acknowledged.

Of course, deliberate substructure in data items is ubiquitous, and it is common practice to document it with the use of data types. Furthermore, typing in a language provides a useful form of redundancy which is susceptible to automated checks of internal consistency.

In PAISLey non-primitive sets can be defined using set union ("A U B"), cross-product ("A x B"), enumeration ("{ 'true', 'false' }"), and parenthesization. The domain and range sets of every function, primitive or not, must be declared (although a function need not have arguments). The domain and range declarations can use arbitrary set expressions. Here are three example declarations:

```
f:  ---> A ;
g:  B x C ---> D U E ;
h:  S ---> T.
```

When a function is applied to arguments, their types must be consistent with the domain declaration of the function. Consistency can be defined with the assistance of type conversion, however, so that the composition "h[g[f]]" is perfectly legal if the definitions:

```
A = B x C ;
S = INTEGERS;
D = { 0, 2, 4, 6, 8 };
E = { 1, 3, 5, 7, 9 }
```

have been made. This notion of typing provides all the documentation and redundancy desirable for engineering goals, without sacrificing any of the flexibility attributable to typelessness. All that it requires is the ability to compare any two set expressions for containment, which is easily done given this particular language of set expressions.

4. QUALMS ABOUT OPERATIONAL REQUIREMENTS

Despite the obvious advantages of operational requirements, one cannot help but have certain reservations about the idea. In this section we examine its apparent disadvantages.

4.1. Encroaching on design

Aren't operational specifications actually design specifications rather than requirements specifications? This question is often prompted by the precision, potential for detail, and executability of operational specifications.

Traditionally, it is said that requirements state what is to be done, and a design states how to do it ([Ross & Schoman 77]). In other words: Properly, requirements specify the functional and performance properties of a system, where both "functional" and performance properties are characteristics of the system as experienced by its environment, but the functional ones are expressable in terms of digital logic, while the performance properties concern such physical concepts as time (see 5.4). Design begins when the resources from which the system is to be constructed are introduced. System design is a matter of managing scarce resources to meet performance goals.

Adopting this definition of the boundary between requirements and design, a requirements specification does not stray into design if and only if it avoids managing resources, either explicitly or implicitly. PAISLey enables the requirements analyst to do this, as illustrated by the requirements for the process-control system, the process structure of which is shown in Figure 8 (process interactions are labeled with the type of information that is transferred).

Here the environment is specified by processes simulating the three machines, the line-printer, and the human operator. (The environment relevant to requirements analysis may extend further than this, including, for instance, the people who use the reports and their intended purposes.) The "machine-monitor" processes tend to their individual machines, reading the sensors and providing immediate local feedback. These processes also pass data along to the "factory-monitor" process, which keeps track of global conditions and alarms the "operator" when necessary. All monitoring processes send selected data to the "database", which answers queries from the "operator" and the "report-generator".

Note that no resources are explictly represented in this specification--all of the internal processes are derived directly from the various system functions. We will now argue that neither the process structure nor the structures inside processes place implicit constraints on resources.

Nowhere is the difference between requirements and design more apparent than in the requirements principle of "sufficient processes"--use as many processes as performance analysis and functional decomposition suggest. The machine monitors are separate from each other because each must synchronize itself with a different (asynchronous) machine, and the factory monitor performs a different function altogether. The report generator is a separate process from the database because the database has real-time response requirements and the report generator does not. It is likely that in the design for this system, all of these processes will be implemented by time-multiplexing a single physical processor. The processor will be a scarce resource, and must be allocated (perhaps using priority interrupts) so that the performance requirements on all processes will be met. Thus partition into processes is a way of expressing relationships having to do with functionality, synchronization, performance, etc., and not resource allocation.

Within processes, we have already discussed the nonconstraining nature of applicative expressions. The handling of states is likewise noncommittal. Consider the "database" process: its state space is "DATABASE", the set of all possible process-control databases, and its successor function is "database-cycle". "Database-cycle" processes one input, be it a data update or query, and produces as its value a new member of "DATABASE", which replaces the old process state at the end of the process step.

There are many ways to update a database. Two of the obvious possibilities are to modify records in place, or to create an entirely new copy and then over-write the old one. The former is more efficient but only works well for relatively static data structures; the latter is more flexible but expensive, and also more amenable to reliability measures--the old database is not destroyed until the new has been successfully completed.

These and intermediate possibilities, differing in their resource allocation and performance characteristics, are equally well subsumed by the abstract mechanism of state replacement.

This view of requirements is elegant and satisfying, but it is not the whole story. Pragmatically, a requirement is any property of the proposed system that is necessary to satisfy the originating organization of the acceptability of the system, and these properties may very well include decisions about resources. Use of a particular computer or software subsystem may be required because the originating organization already owns it, and management insists that it be used. There may even be a requirement that the system must fit into a particular amount of memory; this might be the case, for instance, if the system is a monitor of experimental equipment, and shares facilities with other experiments on a satellite. The amount of memory in the on-board computer allocated to each experiment is an administrative decision which must be made (at least tentatively) before work on developing the individual experiments can begin.

The reality is that a system develops through a hierarchy of decisions, each decision constraining those below it in the hierarchy. No system is developed in a political or economic vacuum, and almost no system performs its function without interfacing with any pre-existing computer system; as a result, some decisions are made prematurely or nonoptimally compared to some theoretical decision procedure based on technical grounds alone. Thus, even though resource decisions are premature at the requirements level, any requirements language which is unable to record them will be terribly fragile, performing adequately only in the most idealized of situations.

PAISLey can record resource decisions because resource structures are like any other structures occurring in digital systems. They can definitely be specified if there is a general model of digital computation, which PAISLey offers. Hardware and software modules, for instance, can be specified as processes, and then included as part of the environment of the proposed system. This is a great strength of the operational approach--the promise of no unpleasant surprises when new applications or economic contexts are encountered.

The ultimate test of whether or not a decision belongs in the requirements is whether or not the system could feasibly be constructed in any other way. This criterion can be illustrated by several features of the AFWET system.

The first requirements problem is that the system is required to use a military standard radio link for communication between the test elements and the contractor-supplied part of the communication network.*** This is a classic example of a (premature) resource requirement, since the requirements should confine themselves to the necessity of (and performance constraints on) communication, and let the contractors determine the best method. But accepting the inevitable, we specify the existing radio link as part of the environment of the proposed system (Figure 9).

The next problem concerns the times to which event messages from test elements (new positions and weapons firings) refer. The central system must know these quite accurately to do meaningful simulation, but it cannot infer the times at which they were sent from the times at which they are received, because delay in the ground communication network is sure to be long and erratic.

There seems to be only one solution to this problem: put timestamps on the messages when they arrive at the radio towers (until that point the delay is small and stable, due to the dedicated radio channels). If true, there is nothing wrong with specifying even this very design-like decision in the requirements (Figure 10).

Note, however, that this operational requirements specification is still quite different from a design for the system. The specification of the "radio-tower" processes will show that they all get current times by requesting them from the "real-time-clock" process, which "ticks" once per process step. The asynchronous interaction point within the radio towers will at first appear as the primitive function:

    current-time: ---> TIME.

The necessary performance requirement will be specified as:

    current-time: "time ---> maximum is 1 microsec".

This means that "current-time" must be evaluated, by interacting with a global clock, in one microsecond or less in all radio towers. The design to meet this requirement will probably involve accurate local clocks which are synchronized via some protocol before each test begins.

Finally, we must consider exactly what is meant by "real-time simulation". Is the simulation event-oriented? Is each incoming message processed as it arrives? What if a late-arriving message contradicts something that has already been computed? (The latter could happen if, in the absence of a recent position message, the system extrapolated the current trajectory of a test element; the late-arriving message could show that it had swerved.)

With existing levels of technology, it is not feasible to build a system which performs simulation of this complexity, in anything close to real time, and backtracks on the basis of belated evidence. Therefore it is not over-constraining any feasible design to put a "no-backtracking" policy into the requirements. Also, the simulation must be oriented toard slices of time rather than events because a shell in flight presents a continuous threat over some period of time. The discipline imposed by the operational approach forces us to understand these issues before specifying the requirements for the central simulation facility, which is done as follows.

The relationship between simulation time and real time is shown in Figure 11. The test is viewed as a sequence of "frames", or snapshots, each containing the positions of all active test elements. The "granularity" g of the simulation is the interval between frames. The granularity must be at least as long as the time it takes to compute a frame, or the simulator will fall increasingly far behind.

Let d be the projected maximum delay in the communication network. Then computed frames can be produced with a steady real-time delay of d + g. In computing the frame for time t, the simulator waits until t + d to make sure it has received all messages sent at t or before (messages arriving later than this will have to be ignored), then computes the frame to be ready at t + d + g. The quantities d and g will be incorporated into performance requirements, of course.

This is carried out by the process structure shown in Figure 12. The simulation is clocked by the "input-buffer" process, which collects messages continually but hands them over to the simulator in batches at intervals of g. Each step of the "simulator" process computes a new frame from the old frame, a batch of messages, and various threat and target models. It also produces a batch of "kill" messages, which are transmitted by the "output-buffer" process.

4.2. Too much precision

There can be little question that specifications written in PAISLey are too precise, and based on too many technical principles, for customers, end users, managers, and other untrained personnel to understand.

At the same time, their rigor can be invaluable to the trained analysts who will write them (this is based on numerous experiences of being confronted by surprise with the vagueness of my own ideas about a system). Informal analysis must always come first, but we have not yet fully exploited the potential of formal languages for expressing approximate or incomplete knowledge and real-world concepts.

There is not really a conflict here, simply because nontechnical people do not have to use the same representations that the analysts do. Analysts can communicate with them using diagrams, simplifications, narrow views, partitions and projections, etc. derived from the current PAISLey specification. The process diagrams (Figures 4,8,9,10,11) are nicely visual and seem fairly intuitive, for instance; it is also likely that dataflow diagrams could be used to show major computations without bothering about timing and control, and that semantic nets, which are used in [Mittermeir 80] and [Yeh & Mittermeir 80] to represent data structures, could be used for the same purpose here.

Among the most popular and successful features of SREM (RSL) and PSL/PSA is that specifications are stored in a database from which a variety of up-to-date reports can be generated automatically. We envision PAISLey's being installed in such a database, and hope that user-oriented reports and diagrams could likewise be produced by tools running on the current specification.

4.3. Interface with data-oriented specification techniques

Other researchers have investigated the problem of requirements for data-processing systems, using as a starting point for their formalisms database languages, i.e. languages originally developed to describe the "conceptual schemas" (abstract, virtual, semantic structures) of databases. The notion that a requirements model should be an explicit representation of the proposed system interacting with its environment has also been derived in this context, but with a completely different type of specification for the model. A philosophy of data-oriented modeling is presented in [Balzer & Goldman 79], while [Yeh et al. 79b], [Roussopoulos 79], and [Mittermeir 80] exemplify it. [Smith & Smith 79] defines a particular data-oriented specification language designed to have all the generality, flexibility, and power needed for complete specification of systems from a data-driven perspective.

It is clear that a data-oriented technique is a more natural way than using PAISLey to develop requirements for data-processing systems. Yet data is a vital part of any system, and cannot be ignored by any requirements technique. It is our purpose here to show that process-oriented PAISLey specifications and data-oriented specifications are both based on the underlying model shown in Figure 13, and can potentially be compatible and even complementary. Then analysts will be free to use either or both (in parallel) as the application and phase of development suggest.

In both approaches there are data items which reflect the state of the relevant part of the environment and the state of the system. The basic relationships among data items are even the same: In data-oriented languages data items are organized into types, and types are related by "generalization" (a type/subtype hierarchy) or "aggregation" (a "component-of" hierarchy). In PAISLey set membership provides typing, set union provides generalization, and cross-product provides aggregation.

Thus if the collection of process states in a PAISLey specification is viewed as a database, it differs from a "normal" database only in having a somewhat restricted structure. Furthermore, the restrictions are tailored to the nature and needs of embedded systems. Specifically: (1) its size is fixed, (2) it is divided into a fixed vector of components (process states), and (3) no item is a component of more than one item.

Restrictions (2) and (3) come about because the specification is to be interpreted as a vector of autonomous distributed parallel computations (the latter prevents one item's being shared between process states). Restriction (1) is based on the philosophy of PAISLey (see 5.1), but its acceptability reflects the nature of embedded systems--because of the close interaction between an embedded system and its environment, the environment is stable rather than transient. Consider, for example, the processes representing test elements in the environment of the AFWET system. Presumably a large and open-ended set of planes, tanks, ships, etc. might eventually be used in tests, which would indicate a large and open-ended set of environment processes. But the requirements document puts a definite limit of 26 on the number of test elements active at any one time, and implies that each is to have a dedicated radio channel. It makes much more sense to construct the model with 26 test-element processes "hard-wired" to their channels, and consider the assignment of test-element processes to physical test elements to be outside the scope of the system.

A nice example of the contrast between data-processing and embedded systems is afforded by an airline reservation system, which has aspects of both. In the requirements model to be used in Section 5, the environment of an airline reservation system consists of nothing but processes representing terminals, of which there are a fixed number. Terminals are the only parts of the environment that are relevant to this communication- and synchronization-oriented PAISLey specification. The data-oriented requirements model in [Yeh _et al_. 79a], on the other hand, interprets the environment of an airline reservation system to consist of entities such as airplanes, passengers, flights, and reservations, because these are the environment entities reflected in the _contents_ of the system's database. Note that a fixed size would not be appropriate here.

To return to Figure 13, in models from both approaches there must be computations which update states and interact with (or cause) other computations. Here the situation is one of complementarity rather than compatibility. Database languages specify data updates and retrievals, sometimes informally, but usually with a syntax based on the predicate calculus. They do not specify explicit concurrency, communication, or control. PAISLey, on the other hand, lends itself to specifications in which data manipulations are left primitive (although the work in [Frankel 79] may lead directly to a marriage of functional and database concepts). We are optimistic about the possibility of defining an interface between the two types of specification so that decisions made in one could be translated into the other.

5. THE PAISLEY LANGUAGE

In this section full details of PAISLey are presented, including a new mechanism for process interactions, and specification of performance requirements. An LALR grammar for PAISLey in BNF form can be found in the Appendix.

5.1. Language philosophy

PAISLey is intended to be simple. In particular, only features which are directly associated with run-time semantics are included.

For production purposes the language must be supported by a system which, in addition to storing specification fragments and collecting them into executable configurations (not to mention providing tools for static analysis and report generation), offers such conveniences as scopes, versions, macros, parameters, libraries, meta-notations, etc. The current frenzy of research on "programming environments" makes it plain that the design of such an environment is not a trivial task, and should probably not be undertaken simultaneously with development of the specification semantics. Specifications prepared using any of the above features would be translated into PAISLey (as currently defined) before interpretation.

Stylistically, PAISLey follows APL in using distinct symbols for distinct operators (but has far fewer of them!). This leads to a concise notation in which essentially all words are user-chosen mnemonics. In this decision and the one above, we apply exactly the same philosophy as [Hoare 78].

One other important principle is that every operational structure must be realizable with a bounded amount of resources (time and space). There is a bounded number of processes, no process state can require an unbounded amount of storage, and no process step can require an unbounded amount of evaluation time.

The purpose of this is performance, i.e. making it possible to design systems which are guaranteed to meet their performance requirements. Clearly if a computational path contains an unbounded loop, or may have to construct a data structure of unbounded size, no guarantee that it meets an absolute time constraint is possible. In PAISLey the only unbounded "structure" is the infinite succession of process steps of each process, and this one exception cannot be avoided.

The static system structures which result from the boundedness principle will greatly facilitate proofs of internal consistency, correctness, and other formal properties.

5.2. Sets, functions, processes, and systems

Statements in PAISLey are delimited by semicolons, and comments are enclosed in double quotation marks.

Names are typed for greater readability. The names of functions are always in lower-case letters, and the names of sets are always in upper-case letters (hyphens and integers may be used in either, but they mus begin with alphabetic strings). Constants are either numbers, or strings enclosed in single quotation marks.

There are four kinds of statement: system declarations, function declarations, set definitions, and function definitions. Since a system is a fixed**** tuple of processes, we use the tuple-construction notation for a *system declaration*. A process is declared using a function application which applies its successor function to an expression evaluating to its initial process state. Thus a system consisting of four processes, three being terminals and the fourth being a shared database, would be declared as:

```
(terminal-1-cycle[blank-display],
 terminal-2-cycle[blank-display],
 terminal-3-cycle[blank-display],
 database-cycle[initial-database]
),
```

where the following domain-range declarations would be appropriate:

```
terminal-1-cycle:  DISPLAY ---> DISPLAY;

blank-display:  ---> DISPLAY;

. . .

database-cycle:  DATABASE ---> DATABASE;

initial-database:  ---> DATABASE.
```

Terminal processes have the contents of the current displays as their process states. Note that there is no explicit naming of processes or systems; this would undoubtedly be added as part of any "environment" facilities.

Function declarations give <u>properties</u> of functions, and may therefore be redundant for nonprimitive (defined) functions--when the properties are also deducible from the function's definition. Declarations of nonprimitive functions can and should be checked for consistency with their definitions. All function declaration statements begin with the function name and a colon; what follows is either a domain-range declaration, of which we have seen many, or a performance property (see 5.4).

Set definitions define set names in terms of set expressions, which use set union, cross-product (which has precedence over union), enumeration, and parenthesization (all shown in 3.4.2). Note that the size of all data structures is bounded, because all tuples (members of sets defined by cross-product) have a bounded number of components.

Function definitions define function names in terms of function expressions, and may use formal parameters, even structured parameter lists, to do so. Here are some possible beginnings for function definition statements:

```
new-func-1 = . . . ;
new-func-2[p] = . . . ;
new-func-3[(p,q)] = . . . ;
new-func-4[(p,(q,(r,s)))] = . . . .
```

Formal parameters have the same syntax as function names; the argument structure must, of course, agree with the function's domain declaration.

Function expressions may use function names, formal parameters, constants, applications of functions to arguments, tuple construction, and conditional selection. Conditional selection (like the LISP "cond") has the syntax "/p1:f1, p2:f2, . . . 'true':fn/", and evaluates to the value of the first functional expression "fi" such that the predicate (Boolean-valued functional expression) "pi" evaluates to "'true'". Note that there is no unbounded iteration, such as would be provided by "while . . . do . . .", nor is recursion allowed. Bounded iteration can be specified using composition. The result is that the number of primitive operations to evaluate any function, including a successor function, can be bounded a priori.

As a simple example, consider the following specification of the successor function of a process representing a CRT terminal:

```
terminal-cycle:  DISPLAY ---> DISPLAY;
terminal-cycle[d] = display[display-and-transact[(d,think-of-request)]].
think-of-request:  ---> REQUEST;
display-and-transact:
    DISPLAY x REQUEST ---> DISPLAY x (RESPONSE U ERROR-MESSAGE);
display-and-transact[(d,r)] = (display[(d,r)],transact[r]);
```

transact: REQUEST ---> RESPONSE U ERROR-MESSAGE;

display: DISPLAY x (REQUEST U RESPONSE U ERROR-MESSAGE) ---> DISPLAY.

The process handles one transaction per process step, reflecting both the request and the response in the display. The primitive function "display" can carry out scrolling or whatever other formatting is desired.

Even aiming for a minimum of conveniences, it is impossible to do without some feature for defining groups of nearly identical items. In PAISLey this is done at all levels using the same index notation, as seen in:

VECTOR = #1..10< x INTEGER >,

which defines members of the set "VECTOR" to be 10-tuples of integers. Index notation always denotes a sequence of the expression in angle-brackets, with the first symbol in the brackets used as the sequence delimiter. The integers after the "#" give the lower and upper bounds of the sequencing count. The only index notation without a delimiter symbol is the one for bounded functional composition (application), which makes "#1 .. 3 < func > [arg]" equivalent to "func[func[func[arg]]]".

In most cases what we want is a group of statements or expressions which differ slightly. This is done by operating on names, which are defined so that "syllables" (alphabetic substrings delimited by hyphens) are semantically meaningful. If the header for an index notation begins with a "syllable" before the "#", any syllable matching it in a name in the repeated expression will be replaced by successive integers from the lower bound to the upper bound. Thus:

BIG-SET = J#1..3< U LITTLE-SET-J >

is equivalent to:

BIG-SET = LITTLE-SET-1 U LITTLE-SET-2 U LITTLE-SET-3,

and the system declaration:

```
(k#0..9999<,terminal-k-cycle[blank-display]>,
 database-cycle[initial-database]
)
```

creates a system with 10,000 terminal processes (an airline reservation system!), where the successor function of the thirteenth one is "terminal-12-cycle".

Index notation can even extend over groups of statements. Suppose we want our 10,000 terminals to be identical, except that some identification must be built into "transact", the primitive function whose elaboration will send to and receive from the central system. This can be done by making slight modifications to the terminal specification already given, as follows:

```
k#0..9999
< ; terminal-k-cycle:  DISPLAY ---> DISPLAY;
    terminal-k-cycle[d] =
    . . .
    display-and-transact[(d,r)] = (display[(d,r)],k-transact[r]);
    k-transact:  RESPONSE ---> RESPONSE U ERROR-MESSAGE;
    . . .
>.
```

5.3. <u>Asynchronous interactions</u>

5.3.1. Definition of exchange functions

Asynchronous interactions between processes are specified using three primitive functions known collectively as "exchange functions". An exchange function carries out two-way point-to-point mutually synchronized communication. It has one argument, which provides a value to be output, and always returns a value which was obtained as input. Thus within the process an exchange function looks like any other primitive function; it has, however, the side-effect of carrying out a process interaction. By making interaction primitives masquerade as functions, we achieve compatibility with applicative notation.

An exchange function whose evaluation has been initiated interacts by "matching" (to be explained) with another pending exchange function. The two <u>exchange</u> arguments and terminate, so that each returns as its value the argument of the other.

Each exchange function has two attributes to be specified, namely a type ("x", "xm", or "xr") and a channel (a user-chosen identifier which has the syntax of a function name). The exchange function with type "x" and channel "chan" is named "x-chan", the exchange function with type "xr" and channel "real-time" is named "xr-real-time", etc. Only exchange functions with the same channel can match with each other.

The "x" is the basic type of exchange function. It can match with any other pending exchange function in its class, including another of type "x". If no other exchange is pending, it will wait until one is. If there are several pending match possibilities, a match will be chosen nondeterministically, with the proviso that there must be no lockout (a situation where a pending exchange waits indefinitely while its match opportunities are given to other, more recently evaluated, exchange functions).

Competitive situations occur in most systems. To enable succinct specification of them we have exchanges of type "xm", which behave exactly like "x"'s except that two "xm"'s in the same class cannot match with each other. They can then compete to match with an exchange of some other type, as the examples will show.

Embedded systems typically have real-time interfaces, especially with the processes in their environments. To specify these we need a third type of exchange function, the "xr", which behaves like the others except that it <u>will not wait</u> to find a match. If evaluation of an "xr" is initiated and there is no other pending exchange in its class, the "xr" terminates immediately without matching, returning its own argument as its value. It is always possible to determine whether or not an "xr" matched by giving it an argument distinct from any that it could obtain by exchanging.

Figure 14(a) shows the possible matches of exchange types within a class. Figure 14(b) (from [Friedman & Filman 80]) shows the derivation of the three types. There must be both fully synchronized primitives ("synchronizing"), and also those which do not synchronize themselves ("free-running"). There must be exchanges which can match with their own kind, and those that compete with their own kind. This makes four possibilities, except that a free-running type which exchanges with its own kind would be impossible, because it would require "matching" two simultaneous, instantaneous events.

5.3.2. Examples of fully synchronized interactions

In this section we will use exchange functions to specify the interactions between transaction-processing terminals and the central database. "Transact" in the terminal specification is elaborated as follows:

```
transact[r] =
     receive-response[send-request[r]];
send-request:  REQUEST ---> FILLER;
send-request[r] = xm-requ[r];
receive-response:  FILLER ---> RESPONSE U ERROR-MESSAGE;
receive-response['null'] = x-resp['null'],
```

and the database process successor function is specified as:

```
database-cycle[d] =
     finalize-transaction[perform-transaction[(d,receive-request)]];

receive-request:  ---> REQUEST;

receive-request = x-requ['null'];

perform-transaction:  DATABASE x REQUEST ---> DATABASE x RESPONSE;

finalize-transaction:  DATABASE x RESPONSE ---> DATABASE;

finalize-transaction[(d,r)] = proj-2-1[(d,send-response[r])];

send-response:  RESPONSE ---> FILLER;

send-response[r] = x-resp[r].
```

By renaming (redefining) the exchange functions with mnemonic names, we are also able to type their domains and ranges. Exchange functions themselves are typeless because they must handle all types.

"Send-request" in a terminal and "receive-request" in the database match with each other to transmit the request. Note that the type "xm"'s in the terminal compete for the type "x" in the database; if nothing but "x"'s were used, two evaluations of "send-request" might match with each other! Since the "xm" and "x" are symmetric with respect to synchronization, either may have to wait for the other.

After the request is processed against the database, "finalize-transaction" disposes of the results. It is defined in terms of the intrinsic function "proj-2-1", which projects an ordered pair onto its first component, in this case the updated database. The second component is evaluated only for its side-effect of sending the response back, and the "'null'" value it returns is thrown away.

"Receive-response" could have been defined using type "xm", but an "x" is also correct, because precedence constraints enforced by the functional nesting of "send-request" inside "receive-response" ensure that at most one instance of "receive-response" will be in evaluation at any one time, namely that of the process whose request is now being processed. Thus matching in the class "resp" is always unique.

5.3.3. Examples of free-running interactions

A "free-running" process is one whose only interactions occur via "xr", so that it will never wait to synchronize with another process. The prototypical free-running process is a real-time clock, which "ticks" once per process step, and could not fulfill its intended function if it had any synchronizing interactions. Such a process is specified:

```
(clock-cycle[0], . . . );
clock-cycle: TIME ---> TIME;
clock-cycle[t] = proj-2-1[(increment[t],offer-time[t])];
increment: TIME ---> TIME;
offer-time: TIME ---> FILLER U TIME;
offer-time[t] = xr-time[t].
```

Any process wishing to read the current time must evaluate:

```
current-time: ---> TIME;
current-time = xm-time['null'].
```

Concurrent "xm-time"'s will compete to match with "xr-time", implying for this particular specification that no two readers will ever get the same clock value.

Another common type of free-running process is a digital simulation of a nondigital, unintelligent environment object. Here is the top-level specification of the processes representing the machines in the environment of a process-control system ([Zave & Yeh 81]):

```
j#1..3
< ; machine-j-cycle: MACHINE-STATE ---> MACHINE-STATE;
    machine-j-cycle[m] =
         proj-2-1[(simulate-machine[(m,feedback-j-if-any)],
                   offer-machine-j-data[sense[m]]
              )];
    feedback-j-if-any: ---> FEEDBACK U FILLER;
    feedback-j-if-any = xr-j-back['null'];
    simulate-machine:
         MACHINE-STATE x (FEEDBACK U FILLER) ---> MACHINE-STATE;
    sense: MACHINE-STATE ---> SENSOR-DATA;
    offer-machine-j-data: SENSOR-DATA ---> FILLER U SENSOR-DATA;
    offer-machine-j-data[s] = xr-j-sens[s]
>.
```

During each process step two things are done in parallel: (1) "simulate-machine" computes the next process state, which is an element of "MACHINE-STATE" encoding the machine's current status, and (2) the current output of sensors attached to the machine ("sense[m]") is offered to the control system via "xr-j-sens". If the control system is ready to accept the data from this machine cycle an exchange will take place; otherwise the data will be gone forever.

"Simulate-machine" has two arguments: the current machine state, and the value returned by "feedback-j-if-any". This function is defined as "xr-j-back", an exchange function which interacts with several sites in the control system which provide controlling feedback to the jth machine. If some actuator is being activated at the moment "xr-j-back" is evaluated, an exchange takes place and a value in "FEEDBACK" is returned. Otherwise the argument "´null´" is returned, indicating that no actuators are being used.

Our final example of a free-running process is a producer-consumer buffer. Its process state is the current buffer contents, and is successor function is:

```
next-buffer:  BUFFER ---> BUFFER;
next-buffer[b] = give-to-consumer[get-from-producer[b]];
get-from-producer:  BUFFER ---> BUFFER;
get-from-producer[b] = /full[b]:  b,
                        ´true´ :  put-on-tail[(xr-prod[´null´],b)]
                       /;
give-to-consumer:  BUFFER ---> BUFFER;
give-to-consumer[b] =
     /empty[b]:  b,
      ´true´  :  put-on-head[(rest[b],xr-cons[first[b]])]
     /.
```

On each process step "get-from-producer" provides the opportunity to put one new element in the buffer (assuming it is not already full). If some producer has a pending "xm-prod[new-element]", "new-element" will be returned as the value of "xr-prod" and inserted. Otherwise "xr-prod" returns "´null´", which "put-on-tail" will simply ignore.

Likewise, on each process step "give-to-consumer" offers the element at the head of the buffer ("first[b]") to any process evaluating "xm-cons[´null´]". If such an evaluation is pending an exchange will take place, and "xr-cons[first[b]]" will return "´null´", which "put-on-head" will ignore. Otherwise the unconsumed "first[b]" will be returned, and "put-on-head" will reinstate it.

The expected behavior of this process (at least under light loading) will be to cycle very fast, checking for interactions but not having any on most process steps. This shows that exchange functions are in some sense more primitive than synchronization mechanisms which enable a process to wait for any one of several events to occur. The payoff is a much simpler implementation for exchange functions, and the choice is in keeping with the PAISLey philosophy of simplicity and minimal semantics. It is also arguable that the above specification is as perspicuous as any, largely because of the benefits of applicative style.

5.3.4. Implementation

In almost all cases the pattern of matches within a class is one-to-one or many-to-one, the latter for resource competition. In this section we present an efficient distributed algorithm for implementing exchange matching in these cases (one additional condition: it cannot be many-to-one matching where the "xr"'s are the "many"). In all cases a central matching facility for each class will do the job.

Consider first an exchange class with many "xm"'s and one "x" (or just two "x"'s, in which case one of them takes the role of the "xm" in this description), all residing at different nodes of a network (this is illustrated in Figure 15). When an "xm" is initiated, a message carrying its argument is sent to the node where the matching "x" resides. These messages are queued up in arrival order. When the "x" is initiated, if the queue is empty, it waits until it is not. When the queue is not empty, it removes the first entry as the "match", takes the value stored there as its own value, sends a termination message containing its argument to the matching "xm", and continues. Computation can continue at the "xm" as soon as the termination message (with its value) is received.

This implementation uses only two messages per match, and automatically prevents lockout with FCFS queueing. For classes with one "xr" and either one "x" or many "xm"'s, the queue is formed at the site of the "xr", and the only modification necessary is that if the "xr" is initiated when the queue of possible matches is empty, then it does not go into the wait state.

5.3.5. Further properties and justifications

Because exchange functions are only "pseudo-functions" and have side-effects, expressions containing them cannot be optimized to avoid evaluation of expressions whose values are not needed. The most common example of this is a successor function with the form "proj-2-1[(a,b)]", where expression "a" computes the next state and "b" interacts with other processes.

There is also a potential problem with distributing values obtained by interaction, but the formal parameter mechanism does this nicely. Suppose the effect of

```
    /equal[(x-denom['null'],0)]:  'divide-check',
     'true':                      divide[(numerator,x-denom['null'])]
    /
```

is wanted, where both usages of the value returned by an exchange are supposed to result from a single evaluation. This can be specified unambiguously by defining "quotient" as:

```
    quotient[(n,d)] = /equal[(d,0)]:  'divide-check',
                       'true'      :  divide[(n,d)]
                      /,
```

and then using it in the invocation "quotient[(numerator,x-denom['null'])]".

Establishing the internal consistency of a specification with exchange functions requires some attention. The range of a user-chosen function defined as an exchange must agree with the domains of all those with which it can exchange. Furthermore, precedence constraints caused by nested evaluation structures can cause exchange deadlocks. But the channel of an exchange function has been made a constant attribute rather than an argument to it just so that exchange patterns would yield to static analysis, and simple arguments do establish deadlock-freedom in many common cases. For instance, the process hierarchy visible in Figure 8 expresses the acyclic "dependency" structure of the interactions in the system; the argument that this prevents deadlock is a common one in the operating system literature (e.g. [Brinch Hansen 77]).

There are so many proposals for distributed interaction mechanisms current today that comparison and justification are essential. Most properly, exchange functions are motivated and justified by our goal of fitting processes and asynchronous interactions into an applicative framework, and in this role they are almost unique (see also [Milne & Milner 79]). Their generality is established by Figure 14(b) and by extensive experience with them, which indicates that the only kind of interaction they cannot specify is unbounded broadcast.

Exchange functions can also be justified, however, on the same basis as procedure-based mechanisms, which fall into the two general categories of procedure-call mechanisms ([Brinch Hansen 78], [Hoare 74], [Ichbiah *et al.* 79]) and message-passing ([Rao 80]). Exchange functions are more primitive than procedure calls because they only specify interaction at one point in time rather than two (procedure call and return). They are thus more general and easier to implement, while the mutual synchronization of the communicating processes provides much of the structure and control usually associated with procedure-call mechanisms.

It is the mutual synchronization that most distinguishes exchange functions from message-passing mechanisms, where (usually) messages are automatically buffered, so that the sender transmits the message and continues, while the message is queued until the receiver is ready for it.

The decision against this scheme is based on our concern with performance. Consider a set of terminals sending updates to a central database. With exchange functions a terminal cannot create new work for the system until the system has accepted its previous work. If a terminal could simply send an update message and continue, its speed could increase (unchecked by the ability of the system to handle the work), the queue at the database could grow to unbounded lengths, and no bounds on the performance of the system could ever be established.

At the same time, there is nothing wrong with <u>bounded</u> buffering, but this can always be specified in PAISLey. But introducing bounds within an abstract, general-purpose interaction mechanism (such as "message passing up to some bound") would seem a most unfortunate mixture of specification and implementation.

Given that synchronization is going to be two-way, it costs very little in the implementation to preserve the possibility of two-way data transfer, although it is seldom used. It also keeps the number of primitives down by a factor of two, since otherwise each of the three exchange functions would have to come in a "sending data" and a "receiving data" version.

Of all the well-known interaction mechanisms, the most similar to exchange functions is Hoare's input/output primitives. In Hoare's language, a pair of statements, "P?input" in process Q and "Q!output" in process P, will come together in the same mutually synchronized manner that two matching exchanges do. "Output" is an expression whose value is assigned to the variable "input", assuming appropriate type correspondences. In addition to the relatively unimportant data asymmetry, Hoare's primitives seem to be different from exchange functions in three fundamental ways: (1) There is no way to specify real-time or free-running interactions. (2) There is no straightforward way to specify resource sharing, since all "matches" are one-to-one by process name. In Hoare's language a process representing a shared resource must have a separate command for each process with which it can communicate, and guard that command ([Dijkstra 75]) with an input command naming the appropriate process of the many. The guard (and statement) to be executed are chosen nondeterministically from the processes that are ready to communicate. These multiple statements seem distinctly clumsy compared to an "xm"/"x" exchange match. Furthermore, the full knowledge each process must have about the names of the processes with which it communicates makes modularity difficult to achieve. (3) Hoare's primitives belong in a procedural, rather than applicative, framework. The destination of a data transfer, for instance, is specified by an address.

5.4. Performance requirements

5.4.1. Definition of performance requirements

So far the only structure that has been needed for complete and formal specification of performance requirements is attachment of timing and reliability attributes to functions in the "functional" requirements specification. A timing attribute refers to the evaluation time of the function. It is a random variable, and any information about its distribution, such as lower or upper bounds, mean, or the distribution itself, may be given.***** Timing attributes for exchange functions are attached to the channel, and hold for all interactions on that channel.

A reliability attribute can only be attached to a function whose range is divided into two subsets (e.g. "---> SUCCESS-RESULT U FAILURE-RESULT"), the first for the values returned by successful evaluations, and the second for values returned when the evaluation fails. The attribute itself is a discrete (binary) random variable whose two outcomes denote successful or failed evaluations, and any information about its distribution may be given. Reliability attributes for exchange functions are attached to the channel, and hold for all interactions on that channel. Furthermore, when an exchange function fails it must match with another whose evaluation also fails, with the values of both being selected at random from the "failure" subsets of their ranges. This restriction is made so that failures will not affect or complicate analysis of exchange patterns. Failure of a nonprimitive function simply means that it delivers a value in the second subset of its range.

Reliability is a difficult and little-understood subject, but this definition of it has several appealing properties. It forces the specified system to have the primary characteristic of a reliable system, namely going into a well-defined and previously anticipated state when something fails. It makes reliability independent of timing and functionality, since a function evaluation must satisfy its timing requirements and deliver a value in the declared range regardless of whether it succeeds or fails. In fact, we have deemed this property so important that we have sacrificed some realism for it: only primitive functions can really fail, since nonprimitive ones are always evaluated according to their definitions. Much more knowledge of reliability is needed before we can be sure how successful this approach will be, but its formality and tractability are strong arguments in its favor.

These performance requirements can be simulated by the specification interpreter, and checked (in principle!) for internal consistency, just as the functional ones are. This means, for instance, that if "f[x]" is defined as "g[h[x]]", and there are upper bounds on the evaluation times of all three, then the upper bound on "f" must be strictly greater than (allowing time for invocation/argument transfer) the sum of the upper bounds on "g" and "h".

5.4.2. Examples of "synchronous closed-loop" performance requirements

An on-line database system can be called a "synchronous closed-loop" system--"closed-loop" because the entire feedback loop realized by the system is explicitly represented, and "synchronous" because the terminal process (on behalf of the cooperative person behind it) waits for responses, i.e. synchronizes itself with the system. For these systems the basic performance requirements are particularly easy to specify, and all are attached to the terminals. We will refer to the functional terminal specification in 5.2.

A response-time limit of 3 seconds is specified by:

```
transact: "time ---> maximum = 3 sec".
```

(Performance requirements are currently just comments in the PAISLey syntax because we have not yet settled on a formal language for distributions.) An average load of 200 transactions per second is specified by:

```
terminal-cycle: "time ---> mean = 50 sec",
```

which says that on the average a terminal demands a transaction (goes through a cycle) every 50 seconds. Finally, the requirement that at least 99 per cent of all transactions must be processed successfully is expressed as:

```
transact: "reliability ---> prob{ 'success' } >= .99",
```

which, of course, can only be attached to "transact" because its range is divided into success ("RESPONSE") and failure ("ERROR-MESSAGE") subranges.

5.4.3. Examples of "asynchronous closed-loop" performance requirements

The process-control system depicted in Figure 8 can be called an "asynchronous closed-loop" system--"asynchronous" because the machines, which are the source and destination of the major feedback loop realized by the system, are free-running. The system must keep up with them without their cooperation. Performance requirements for these systems are more of a challenge, but the operational approach enables us to specify them straightforwardly. "Open-loop" specifications, in which not all of the feedback loop (ultimately, the purpose of <u>any</u> embedded system is to realize feedback loops) is included explicitly in the model, have performance requirements similar to these. An example of an open-loop specification would be a patient-monitoring system in which treatment of patients was not represented, only display of warning messages.

We will now present the timing requirements for the process-control system. The machine processes specified in 5.3.3 were designed to carry out a fixed-interval simulation with step time or granularity .1 second. This is specified:

```
j#1..3< ; machine-j-cycle:  "time ---> = .1 sec" >.
```

Recall from 4.1 that there are two closed feedback loops realized by this system. The fully automatic feedback loop for conditions local to individual machines is provided by the "<u>machine-monitor</u>" processes. The partially manual feedback loop for dangerous factory conditions includes the "<u>machine-monitor</u>" processes, the "<u>factory-monitor</u>" process, and the "<u>operator</u>" process directly in its realization. Factory engineers give these two loops response-time limits of 3 and 60 seconds, respectively.

The automatic feedback loop will be considered first. The successor functions of the machine monitors are defined as follows:

```
j#1..3
< ; machine-j-monitor-cycle:  MACHINE-IMAGE ---> MACHINE-IMAGE;

    machine-j-monitor-cycle[m] =
         process-machine-j-data[(m,get-machine-j-data)];

    get-machine-j-data:  ---> SENSOR-DATA;

    get-machine-j-data = x-j-sens['null'];

    process-machine-j-data:
         MACHINE-IMAGE x SENSOR-DATA ---> MACHINE-IMAGE;

    process-machine-j-data[(m,d)] =
         proj-3-1[(maintain-machine-image[(m,d)],
                   feedback-j-if-needed[check-machine-condition[(m,d)]],
                    provide-machine-j-data[(m,d)]
                   )];

          maintain-machine-image:
               MACHINE-IMAGE x SENSOR-DATA ---> MACHINE-IMAGE;

          check-machine-condition:
               MACHINE-IMAGE x SENSOR-DATA ---> FEEDBACK U { 'ok' };

          feedback-j-if-needed:  FEEDBACK U { 'ok' } ---> FILLER;

          feedback-j-if-needed[f] = /equal[(f,'ok')]:  'null',
                                     'true'         :  feedback-j[f]
                                    /;

          feedback-j:  FEEDBACK ---> FILLER;

          feedback-j[f] = xm-j-back[f];

          provide-machine-j-data:  MACHINE-IMAGE x SENSOR-DATA ---> FILLER

      >.
```

A monitor begins its step by getting sensor data from its machine (see 5.3.3). "Process-machine-j-data" does three things in parallel with that information: (1) update an "image" of the machine which is kept in the process state for the purpose of making history-sensitive decisions, (2) check to see if feedback is needed and if so provide it, and (3) offer edited forms of the sensor data to other parts of the system. Note that the automatic feedback loop is completely contained within one cycle of a machine monitoring process. This means that the necessary formal performance requirement is simply:

j#1..3< ; machine-j-monitor-cycle: "time ---> maximum = 3 sec" >.

Since the other feedback loop involves action by the environment (the operator) as well as the system, performance allocation of the 60-second leeway must be included in the requirements. Performance allocation is normally a design activity, but this is a typical example of the frequent need to handle "design-like" decisions at the requirements level. If the operator is allocated 50 seconds to respond to the alarm, this decision can be documented by specifying:

operator-cycle: "time ---> maximum = 50 sec",

since the operator's response to an alarm is completely contained within one cycle of that process.

The "factory-monitor" process is very similar to the "machine-monitor" processes, except that it gets its data from the machine monitors instead of the machines, and responds to detecting an undesirable condition by notifying the operator instead of interacting with the machines. Thus the automatic part of this feedback loop is completely contained within one cycle of the "factory-monitor" process, with the understanding that the data it receives from the machine monitors may already be as much as 3 seconds old. The obvious conclusion is that the factory monitor must complete its cycle within 60 - 50 - 3 = 7 seconds:

factory-monitor-cycle: "time ---> maximum = 7 sec".

Both the machine monitors and the operator need to access the database during their cycles, and therefore depend on database response to meet their own performance requirements. Although it is not necessary until the design phase, we can derive a performance requirement for the database that will guarantee adequate service, assuming an implementation with FCFS scheduling, such as that in 5.3.4. (We know of no reasonable method besides FCFS queueing for preventing lockout.) Let us say that every process must be guaranteed a database response/access time of 2 seconds, which we judge will enable both machine monitors and the operator to satisfy their other constraints. Since interactions are mutually synchronized, no process can go on to create more work for the database until its previous request has been processed. This means that the maximum number of outstanding requests is five (five processes have access to the database), and a time limit of .4 seconds will guarantee that all are honored within 2 seconds:

database-cycle: "time ---> maximum = .4 sec".

5.4.4. "Real-world" properties

Time and reliability (the fact that sometimes digital components do not do what their definition says they will, for physical reasons forever beyond the reach of digital logic) are nondigital properties that incontrovertibly affect the digital domain. In [Zave 80b] many other such physical ("real-world") properties are mentioned, weight and distance, for example. Why aren't these performance requirements as well?

The answer is that, to the extent that we know them, the effects of these properties on the computational (digital) domain can be specified in terms of functions, timing, and reliability. Weight constraints, for instance, only affect how many functions can be realized. Even if we did attach weight attributes to components of a PAISLey specification, there is nothing that an interpreter could do with them. Therefore an informal comment is just as satisfactory.

Distance is a more interesting example because its effects on the computational domain are more varied. Distance increases the relative time for interprocess interactions, decreases component reliability, and increases the logical complexity of interfaces which must cope with these factors. Yet these three effects are directly expressable in terms of timing, reliability, and functional requirements, respectively.

Factors such as these can have a profound effect on requirements. In an airline reservation system, for instance, it may be necessary to divide the response-time or transaction-reliability allowances into portions for the data-communication subsystem and portions for the database subsystem. Although (as mentioned before) allocation is technically a design decision, two additional reasons, both applicable here, for doing it during the requirements phase are: (1) to enable feasibility analyses of two very different technologies, and (2) to contract the work to different organizations.

These allocated requirements can be specified in PAISLey. We have constructed a requirements model in which time limits are given for, and failures can occur in, each of three stages: input transmission, transaction processing, and output transmission. Failure at any stage aborts subsequent stages and propagates an appropriate error message. This is the source of elements in the set "ERROR-MESSAGE" found in the range of "transact" in the terminal specification.

6. INTERIM EVALUATION

How well do PAISLey specifications meet the goals of 1.2? Requirements written in this language are certainly precise, unambiguous, and executable, and can be determined to be internally consistent. We have argued that the language allows, and perhaps even encourages, specifications to be modifiable, intuitive, and minimal.

Experience has indicated that PAISLey allows complete specification of requirements properties relevant to the computational domain. It says nothing about constraints on the development process itself, such as deadlines, cost limits, methodological standards, and routine maintainance procedures. It is also not particularly helpful in posing alternate or prioritized requirements ([Yeh et al. 80]). And the need to supplement formal requirements with diagrams, comments, and other informal avenues of human communication will never disappear.

PAISLey also enables nontrivial decomposition of complexity in all three ways. The division of a specification into processes is an especially useful form of partition, because it decomposes both static and dynamic properties, and because it correlates fairly well with our abstract, intuitive notion of system "functions". The partitioning even extends to execution of specifications, because any subset of processes can be executed in isolation, simply by leaving all interactions with missing processes as unelaborated primitives in a form such as "receive-message: ---> MESSAGE". The interpreter will evaluate this "interaction site" by choosing some message at random. This capability was used in [Zave & Yeh 81] to develop a specification in five versions, each independently executable, and each obtained from the last by adding new processes/functions in an "outside-in" sequence.

A projection decomposes complexity by representing only a subset of the system's properties. Performance attributes are defined so that the functional specification is independent of its performance properties, and timing and reliability are independent of each other, both very useful forms of projection. Furthermore, by elaborating a specification only until all process interactions and control-oriented functions are explicit (a most natural thing to do in PAISLey!), and by then specifying the primitive sets and data-manipulation functions in a data-oriented specification language, the analyst can achieve an almost perfect projection of his underlying model onto process-oriented and data-oriented views.

Within processes, state replacement (rather than assignment) and applicative notation offer unsurpassed opportunities for <u>abstraction</u>. Applicative languages actually force the user to create an abstraction/elaboration hierarchy, while the high-level, but procedural, languages now being proposed as design notations continually disrupt it with assignment statements. Processes, however, do not lend themselves so readily to hierarchical representations. More research is needed in this area (see 7.2, where <u>formal manipulability</u> is also mentioned).

7. PLANS FOR FUTURE RESEARCH

7.1. Experience

We have plans to implement a specification interpreter with simple consistency-checking, so as to gain experience with the impact of executable specifications on the requirements development process. This will include consideration of the language front-end, and investigation of the display, report, and trace facilities needed for the results of execution.

7.2. Methodology

This work on requirements specification, which has been pursued so far with small examples, must be extended in the directions of requirements analysis, and "scaling-up" to large systems. Both problems will be attacked by looking for an abstraction methodology for process-based specifications, i.e. a technique for conceiving of and specifying a system of interacting processes as a top-down hierarchically structured set of specifications. The technique for developing a top-down hierarchy would provide a trial analysis methodology, and the existence of the hierarchy would assist analysts in handling the complexity of large systems.

There are several precedents to follow in this effort. One is the well-known arrangement of processes themselves in a hierarchical structure where processes at higher levels "give work to" processes at lower levels ([Parnas 74]); this will be helpful for requirements if and only if the hierarchies so created coincide with some rational hierarchy of system "functions", seen from the requirements viewpoint. Another precedent is the aggregation of related processes into "subsystems", as in DDN ([Riddle et al. 78]). There is clearly some correlation between system "functions" (requirements view) and processes in our example specifications, which is an encouraging sign; furthermore, many of the same patterns of processes are observed over and over again in embedded systems.

One other possibility is the use of purely applicative notation as it is applied in [Smoliar 79] and [Friedman & Wise 79], i.e. to specify parallel and distributed system concepts in a way that is more abstract than process-based notation could be, because of the total lack of states (the concomitant disadvantage is inability to deal with performance or free-running interfaces, see [Zave 80a]). By establishing some formal equivalences between these two notations, we may be able to exploit some of the formal manipulability and power of abstraction characteristic of applicative languages for our own, process-oriented, purposes.

7.3. Design

It has been pointed out that PAISLey is capable of specifying the results of design decisions. A logical extension of this is to investigate its properties as a design specification language. The benefits are potentially great, because a uniform language for requirements and design should make possible substantial improvements in the traceability and automatability of design. It might also lead to a better theoretical understanding of design decisions as resource/performance trade-offs.

8. CONCLUSION

It would not be seemly to end a paper as long as this without a conclusion, but there is little left to say.

In addition to the varied, but small, examples discussed here, PAISLey has been used to specify (in some 33 pages) a distributed design for an innovative interactive numerical system, and the system has been implemented directly from that specification. Throughout the project the specification has served successfully as an interface between the numerical and distributed-system domains, both for human communication and for executable code ([Zave & Rheinboldt 79], [Zave 78], [Zave & Cole 81]).

ACKNOWLEDGMENTS

Many people have contributed to the work presented here. My thanks especially to Bob Fitzwater, with whom the foundations for PAISLey were laid, to Steve Smoliar, Alex Conn, and Roland Mittermeir, for stimulating discussions on requirements, to Priscilla Fowler, for the chance to try these ideas on a class at Bell Labs, to George Cole, for diligent and able assistance, to Dick Hamlet, who would read this stuff before anyone else would (or could), and to Raymond Yeh, for support, encouragement, ideas, and good advice.

REFERENCES

[Air Force 65]
U.S. Air Force, "Air Force Weapons Effectiveness Testing (AFWET) Instrumentation System", R&D Exhibit No. PGVE 64-40, Air Proving Ground Center, Eglin Air Force Base, Florida, 1965.

[Alford 77]
Mack W. Alford, "A Requirements Engineering Methodology for Real-Time Processing Requirements", IEEE Trans. Software Engr. SE-3, January 1977, pp. 60-69.

[Backus 78]
John Backus, "Can Programming be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs", Comm. ACM 21, August 1978, pp. 613-641.

[Balzer & Goldman 79]
Robert Balzer and Neil Goldman, "Principles of Good Software Specification and Their Implications for Specification Language", Proc. Specifications of Reliable Software Conf., Cambridge, Mass., April 1979, pp. 58-67.

[Belady & Lehman 79]
L.A. Belady and M.M. Lehman, "The Characteristics of Large Systems", Research Directions in Software Technology, Peter Wegner, ed., M.I.T. Press, Cambridge, Mass., 1979, pp. 106-138.

[Bell et al. 77]
Thomas E. Bell, David C. Bixler, and Margaret E. Dyer, "An Extendable Appraoch to Computer-Aided Software Requirements Engineering", IEEE Trans. Software Engr. SE-3, January 1977, pp. 49-60.

[Bell & Thayer 76]
T.E. Bell and T.A. Thayer, "Software Requirements: Are They Really a Problem?", Proc. 2nd Intl. Conf. on Software Engineering, San Francisco, Cal., October 1976, pp. 61-68.

[Boehm 76]
Barry W. Boehm, "Software Engineering", IEEE Trans. Computers C-25, December 1976, pp. 1226-1241.

[Brinch Hansen 77]
Per Brinch Hansen, The Architecture of Concurrent Programs, Prentice-Hall, Inc., 1977.

[Brinch Hansen 78]
Per Brinch Hansen, "Distributed Processes: A Concurrent Programming Concept", Comm. ACM 21, November 1978, pp. 934-941.

[Conn 80]
Alex Paul Conn, "Maintenance: A Key Element in Computer Requirements Definition", Proc. COMPSAC '80, Chicago, Ill., October 1980, pp. 401-406.

[Davis & Rauscher 79]
Alan M. Davis and Tomlinson G. Rauscher, "Formal Techniques and Automatic Processing to Ensure Correctness in Requirements Specifications", Proc. Specifications of Reliable Software Conf., Cambridge, Mass., April 1979, pp. 15-35.

[Davis & Vick 77]
Carl G. Davis and Charles R. Vick, "The Software Development System", IEEE Trans. Software Engr. SE-3, January 1977, pp. 69-84.

[Dijkstra 75]
E.W. Dijkstra, "Guarded Commands, Nondeterminacy, and Formal Derivation of Programs", Comm. ACM 18, August 1975, pp. 453-457.

[Fisher 78]
David A. Fisher, "DoD's Common Programming Language Effort", Computer 11, March 1978, pp. 24-33.

[Filman & Friedman 80]
Robert E. Filman and Daniel P. Friedman, Languages and Models for Distributed Computing, to appear.

[Fitzwater & Zave 77]
D.R. Fitzwater and Pamela Zave, "The Use of Formal Asynchronous Process Specifications in a System Development Process", Proc. 6th Texas Conf. on Computing Systems, Austin, Texas, November 1977, pp. 2B-21 - 2B-30.

[Frankel 79]
R.E. Frankel, "FQL--The Design and Implementation of a Functional Database Query Language", Univ. of Penn. Decision Sciences 79-05-13, Philadelphia, Penn., 1979.

[Friedman & Wise 77]
Daniel P. Friedman and David S. Wise, "Aspects of Applicative Programming for File Systems", Proc. ACM Conf. on Language Design for Reliable Software, Raleigh, N. Car., March 1977, pp. 41-55.

[Friedman & Wise 78a]
Daniel P. Friedman and David S. Wise, "Aspects of Applicative Programming for Parallel Processing", IEEE Trans. Computers C-27, April 1978, pp. 289-296.

[Friedman & Wise 78b]
Daniel P. Friedman & David S. Wise, "Unbounded Computational Structures", Software--Practice and Experience 8, July-August 1978, pp. 407-416.

[Friedman & Wise 79]
Daniel P. Friedman and David S. Wise, "An Approach to Fair Applicative Multiprogramming", Semantics of Concurrent Computation (G. Kahn, ed.), Lecture Notes in Computer Science 70, Springer-Verlag, Berlin, 1979, pp. 203-226.

[Friedman & Wise 80]
Daniel P. Friedman and David S. Wise, "An Indeterminate Constructor for Applicative Programming", Proc. 7th Annual ACM Symp. on Principles of Programming Languages, Las Vegas, Nev., January 1980, pp. 245-250.

[Heninger 79]
Kathryn L. Heninger, "Specifying Software Requirements for Complex Systems: New Techniques and Their Application", Proc. Specifications of Reliable Software Conf., Cambridge, Mass., April 1979, pp. 1-14.

[Hoare 74]
C.A.R. Hoare, "Monitors: An Operating System Structuring Concept", Comm.

ACM 17, October 1974, pp. 549-557.

[Hoare 78]
C.A.R. Hoare, "Communicating Sequential Processes", Comm. ACM 21, August 1978, pp. 666-677.

[Horning & Randell 73]
J.J. Horning and B. Randell, "Process Structuring", Computing Surveys 5, March 1973, pp. 5-30.

[Ichbiah et al. 79]
J.D. Ichbiah et al., "Rationale for the Design of the Ada Programming Language", SIGPLAN Notices 14, June 1979, Part B.

[Ingalls 78]
Daniel H.H. Ingalls, "The Smalltalk-76 Programming System Design and Implementation", Proc. 5th Annual ACM Symp. on Principles of Programming Languages, Tucson, Ariz., January 1978, pp. 9-16.

[Iverson 80]
Kenneth E. Iversion, "Notation as a Tool of Thought", Comm. ACM 23, August 1980, pp. 444-465.

[Knight 72]
John R. Knight, "A Case Study: Airlines Reservations Systems", Proc. of the IEEE 60, November 1972, pp. 1423-1431.

[Mao & Yeh 80]
William T. Mao and Raymond T. Yeh, "Communication Port: A Language Concept for Concurrent Programming", IEEE Trans. Software Engr. SE-6, March 1980, pp. 194-204.

[Milne & Milner 79]
George Milne and Robin Milner, "Concurrent Processes and Their Syntax", Jour. ACM 26, April 1979, pp. 302-321.

[Mittermeir 80]
Roland T. Mittermeir, "Semantic Nets for Modeling the Requirements of Evolvable Systems--An Example", Institut fuer Digitale Anlagen, Technische Universitaet Wien, Vienna, Austria, May 1980.

[Parnas 74]
David L. Parnas, "On a 'Buzzword': Hierarchical Structure", Proc. IFIP Congress, Stockholm, Sweden, 1974.

[Rao 80]
Ram Rao, "Design and Evaluation of Distributed Communication Primitives", Univ. of Wash. Computer Science 80-04-01, Seattle, Wash., April 1980.

[Riddle et al. 78]
William E. Riddle et al., "Behavior Modeling During Software Design", IEEE Trans. Software Engr. SE-4, July 1978, pp. 283-292.

[Ross 77]
Douglas T. Ross, "Structured Analysis (SA): A Language for Communicating Ideas", IEEE Trans. Software Engr. SE-3, January 1977, pp. 16-34.

[Ross & Schoman 77]
Douglas T. Ross and Kenneth R. Schoman, "Structured Analysis for Requirements Definition", IEEE Trans. Software Engr. SE-3, January 1977, pp. 6-15.

[Roussopoulos 79]
Nicholas Roussopoulos, "CSDL: A Conceptual Schema Definition Language for the Design of Data Base Applications", IEEE Trans. Software Engr. SE-5, September 1979, pp. 481-496.

[Smith & Smith 79]
John Miles Smith and Diane C.P. Smith, "A Data Base Approach to Software Specification", Proc. Software Development Tools Workshop, Pingree Park, Colo., May 1979, (Springer-Verlag, W.E. Riddle and R.E. Fairley, eds., 1980), pp. 176-200.

[Smoliar 79]
Stephen W. Smoliar, "Using Applicative Techniques to Design Distributed Systems", Proc. Specifications of Reliable Software Conf., Cambridge, Mass., April 1979, pp. 150-161.

[Smoliar 80]
Stephen W. Smoliar, "Applicative and Functional Programming", Software Engineering Handbook, C.V. Ramamoorthy and C.R. Vick, eds., Prentice-Hall, Inc., to appear.

[Teichroew & Hershey 77]
Daniel Teichroew and Ernest A. Hershey III, "PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems", IEEE Trans. Software Engr. SE-3, January 1977, pp. 41-48.

[Yeh et al. 79a]
Raymond T. Yeh et al., "Software Requirement Engineering--A Perspective", Univ. of Texas Computer Science SDBEG-7, Austin, Texas, March 1979.

[Yeh et al. 79b]
Raymond T. Yeh, Nick Roussopoulos, and Philip Chang, "Systematic Derivation of Software Requirements Through Structured Analysis", Univ. of Texas Computer Science SDBEG-15, Austin, Texas, 1979.

[Yeh et al. 80]
Raymond T. Yeh et al., "Software Requirements: A Report on the State of the Art", Univ. of Maryland Computer Science TR-949, College Park, Maryland, October 1980 (to appear as "Software Requirements: New Directions and Perspectives" in Software Engineering Handbook, C.V. Ramamoorthy and C.R. Vick, eds., Prentice-Hall, Inc.).

[Yeh & Mittermeir 80]
Raymond T. Yeh and Roland T. Mittermeir, "Conceptual Modeling as a Basis for Deriving Software Requirements", Proc. Intl. Computer Symp., Taipei, Taiwan, December 1980, to appear.

[Zave 78]
Pamela Zave, "The Formal Specification of an Adaptive, Parallel Finite-Element System", Univ. of Maryland TR-715, College Park, Maryland, December 1978.

[Zave 80a]
Pamela Zave, "Applicative Specifications of Distributed Systems: Extending them to Embedded Systems", submitted for publication, 1980.

[Zave 80b]
Pamela Zave, "'Real-World' Properties in the Requirements for Embedded Systems", Proc. 19th Annual Wash., D.C. ACM Tech. Symp., Gaithersburg, Md., June 1980, pp. 21-26.

[Zave & Cole 81]
Pamela Zave and George E. Cole, Jr., "A Quantitative Evaluation of the Feasibility of, and Suitable Hardware Architectures for, an Adaptive, Parallel Finite-Element System", in preparation.

[Zave & Rheinboldt 79]
Pamela Zave and Werner C. Rheinboldt, "Design of an Adaptive, Parallel Finite-Element System", ACM Trans. Math. Software 5, March 1979, pp. 1-17.

[Zave & Yeh 81]
Pamela Zave and Raymond T. Yeh, "Executable Requirements for Embedded Systems", Proc. 5th Intl. Conf. on Software Engr., San Diego, Cal., March 1981, to appear.

APPENDIX: A GRAMMAR FOR PAISLEY

This grammar is LALR, and is written in BNF with nonterminals underlined.

Comments are transparent, and can therefore appear anywhere. Blanks are also transparent, except inside an ascii-string.

```
comment ::= "ascii-string"
--------------------------------------------------------------------------

spec ::= spec ; statement |
         statement |
         spec ; index-head < ; spec > |
         index-head < ; spec >

index-head ::= lower-string # integer .. integer |
               upper-string # integer .. integer |
               # integer .. integer

statement ::= system-decl |
              func-decl |
              set-defn |
              func-defn
--------------------------------------------------------------------------

system-decl ::= ( process-list )

process-list ::= process-list , process |
                 process |
                 process-list , index-head < , process-list > |
                 index-head < , process-list >

process ::= func-name [ func-exp ]
--------------------------------------------------------------------------

func-decl ::= func-name : func-property

func-property ::= domain-range |
                  timing-attribute |
                  reliability-attribute

domain-range ::= set-exp ---> set-exp |
                 ---> set-exp

timing-attribute ::= comment

reliability-attribute ::= comment
--------------------------------------------------------------------------

set-defn ::= set-name = set-exp
```

```
set-exp ::= set-exp U set-term |
            set-term |
            set-exp U index-head < U set-exp > |
            index-head < U set-exp >
set-term ::= set-term x set-item |
             set-item |
             set-term x index-head < x set-term > |
             index-head < x set-term >
set-item ::= set-name |
             ( set-exp ) |
             { const-list }
set-name ::= upper-string - set-name-string |
             upper-string
set-name-string ::= set-name-string - set-syll |
                    set-syll
set-syll ::= upper-string |
             integer
const-list ::= const-list , const-name |
               const-name
const-name ::= 'ascii-string' |
               integer |
               real-number
------------------------------------------------------------------------------
func-defn ::= func-name = func-exp |
              func-name formal-params = func-exp
formal-params ::= [ param-list ]
param-list ::= param-list , func-name |
               func-name |
               param-list , ( param-list ) |
               ( param-list )
func-exp ::= func-name |
             const-name |
             func-appl |
             ( func-list ) |
             / pred-pair-list , 'true' : func-exp /
func-appl ::= func-name [ func-exp ] |
              index-head < func-name > [ func-exp ]
func-list ::= func-list , func-exp |
              func-exp |
              func-list , index-head < , func-list > |
              index-head < , func-list >
pred-pair-list ::= pred-pair-list , pred-pair |
                   pred-pair |
                   pred-pair-list , index-head < , pred-pair-list > |
                   index-head < , pred-pair-list >
pred-pair ::= func-exp : func-exp
func-name ::= lower-string |
              lower-string - func-name-string
func-name-string ::= func-syll - func-name-string |
                     func-syll
func-syll ::= lower-string |
              integer
```

Primitives of the grammar.

ascii-string ::= any string of ASCII characters

upper-string ::= any string of upper-case alphabetical characters (note that 'U' is also an operator, and should not be generated as a set-name)

lower-string ::= any string of lower-case alphabetical characters (note that 'x' is also an operator, and should not be generated as a func-name)

integer ::= any string of numerals

real-number ::= any string of numerals with a single embedded period

---

Intrinsic sets.

FILLER = { 'null' }

BOOLEAN = { 'true', 'false' }

INTEGER = the set of all integers representable on the host machine

REAL = the set of all real numbers representable on the host machine

STRING = the set of all string constants with length less than or equal to some bound

---

Typeless intrinsic functions.

x-lower-string | xm-lower-string | xr-lower-string

proj-integer-integer

equal

Typed intrinsic functions.

sum: INTEGER x INTEGER --> INTEGER

difference: INTEGER x INTEGER --> INTEGER

product: INTEGER x INTEGER --> INTEGER

quotient: INTEGER x INTEGER --> INTEGER

remainder: INTEGER x INTEGER --> INTEGER

greater-than: INTEGER x INTEGER --> BOOLEAN

less-than: INTEGER x INTEGER --> BOOLEAN

greater-than-or-equal: INTEGER x INTEGER --> BOOLEAN

less-than-or-equal: INTEGER x INTEGER --> BOOLEAN

FOOTNOTES

*Thus "embedded" is almost synonymous with "real-time", but we prefer the newer term because it does not exclude performance requirements dealing with reliability.

**Throughout this paper mappings will be called "functions", despite the fact that mappings named in specifications are often relations. The reason is that "function" gives a more accurate impression: the intention is always to produce a unique value when the mapping is invoked in the eventual target system, even though that value cannot always be determined by a known functional expression.

***This is actually an inference from the requirements document, which is by no means clear on this point.

****For interpretable languages, "fixed" and "bounded" always mean the same thing, because the programmer declares structures sized up to the bound, and then uses as much of them as needed.

*****More generally, the sequence of evaluations of the function over the lifetime of the system could be associated with a stochastic process, so that the time of each evaluation would be a separate random variable, but let us hope such generality will never be needed.

FIGURES

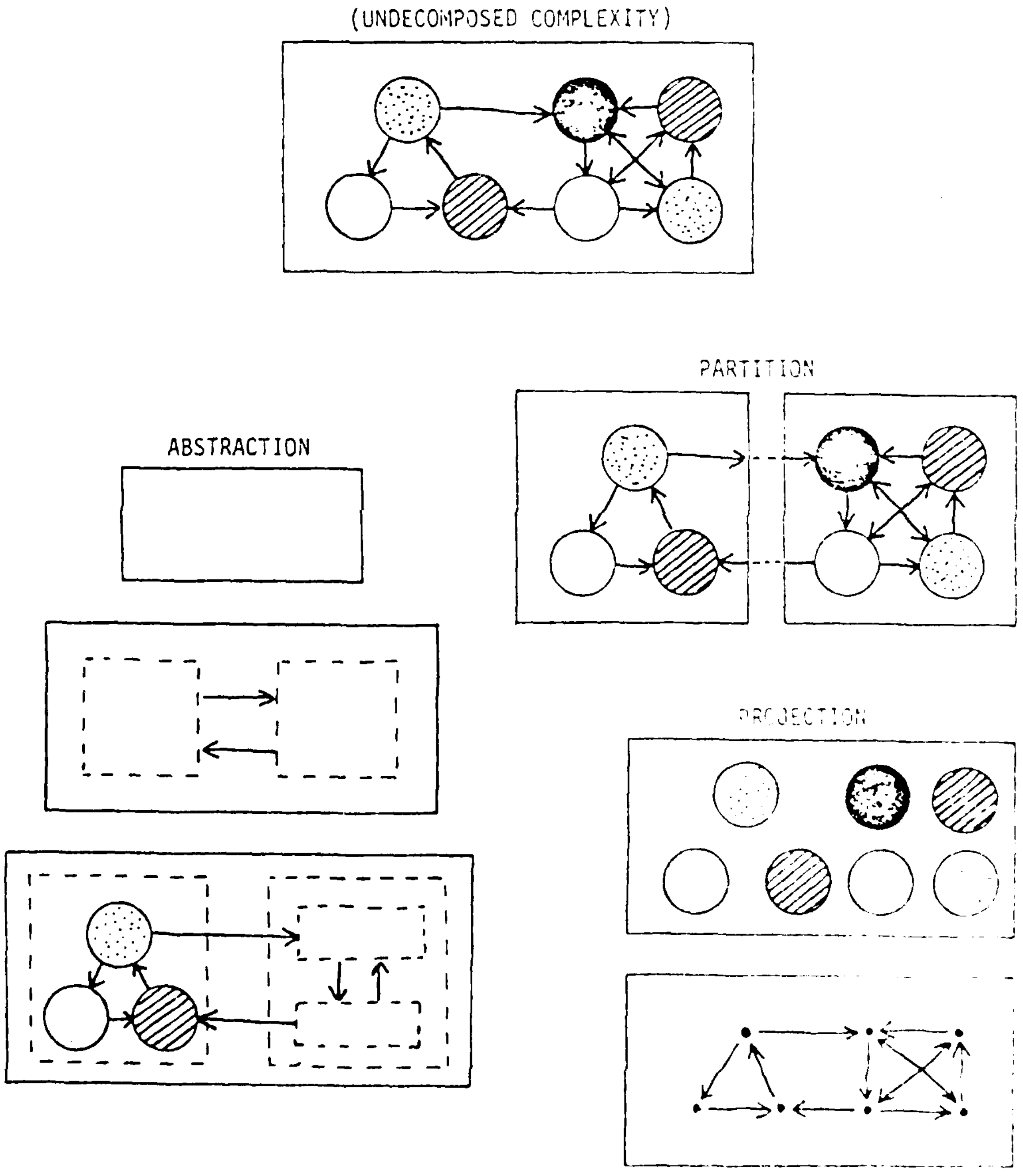


Figure 1. Three ways to decompose complexity.

TYPE	CHARACTERISTICS	EXAMPLES
embedded system	special-purpose (application) absolute performance requirements	industrial process-control system flight-guidance system
data-processing system	special-purpose (application) relative performance requirements	batch business program on-line database system
support system	general-purpose relative performance requirements	operating system software development tool

Figure 2. A requirements-level system classification.

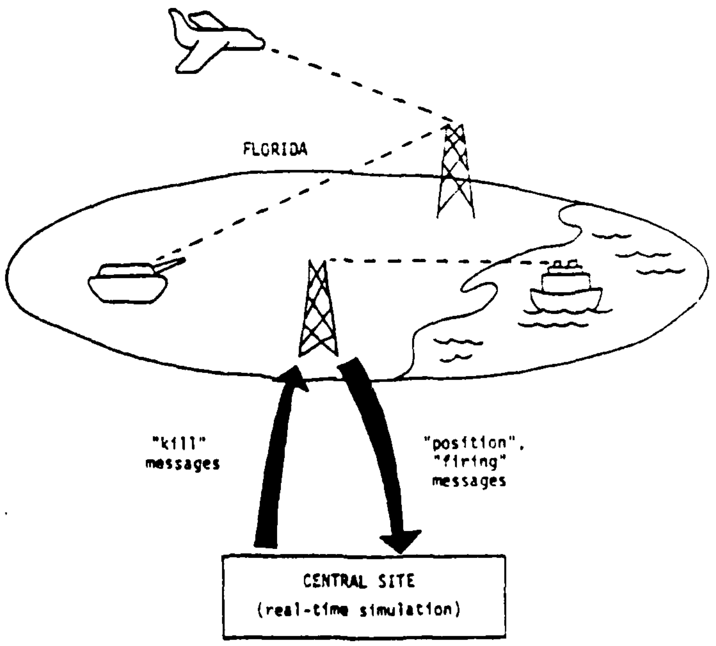


Figure 3. The AFWET system.

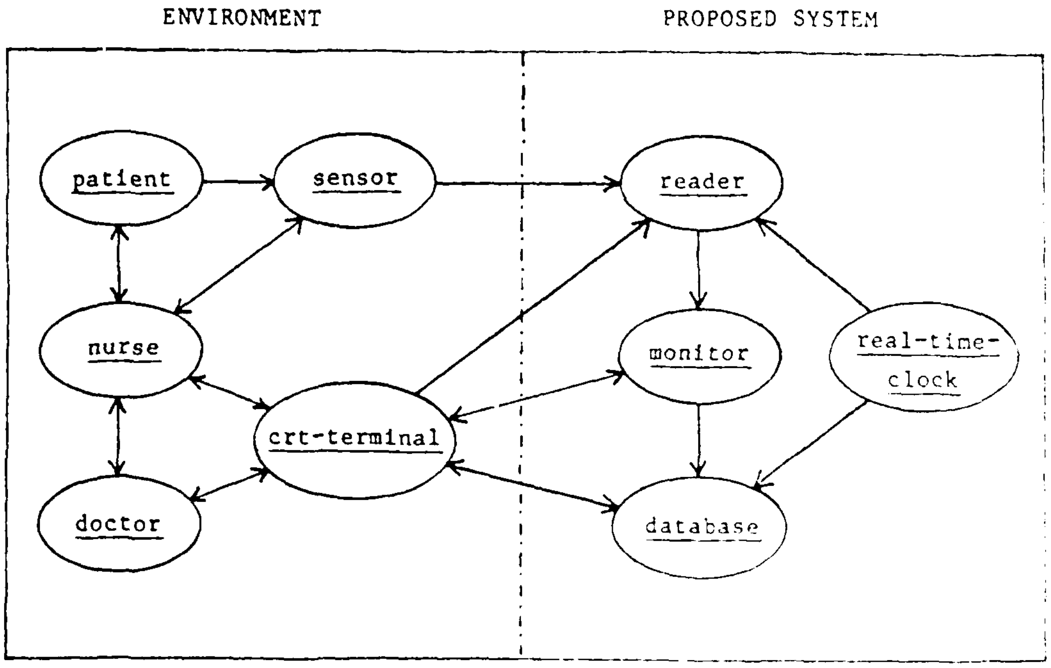


Figure 4. Partial model of a patient-monitoring system.

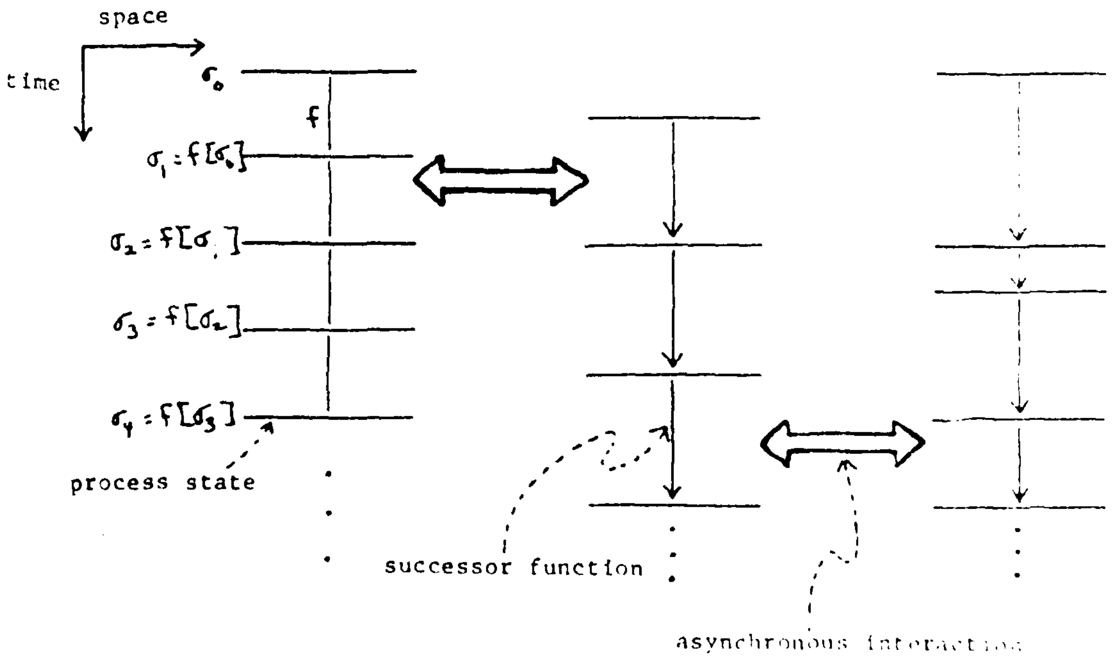


Figure 5. Processes in action.

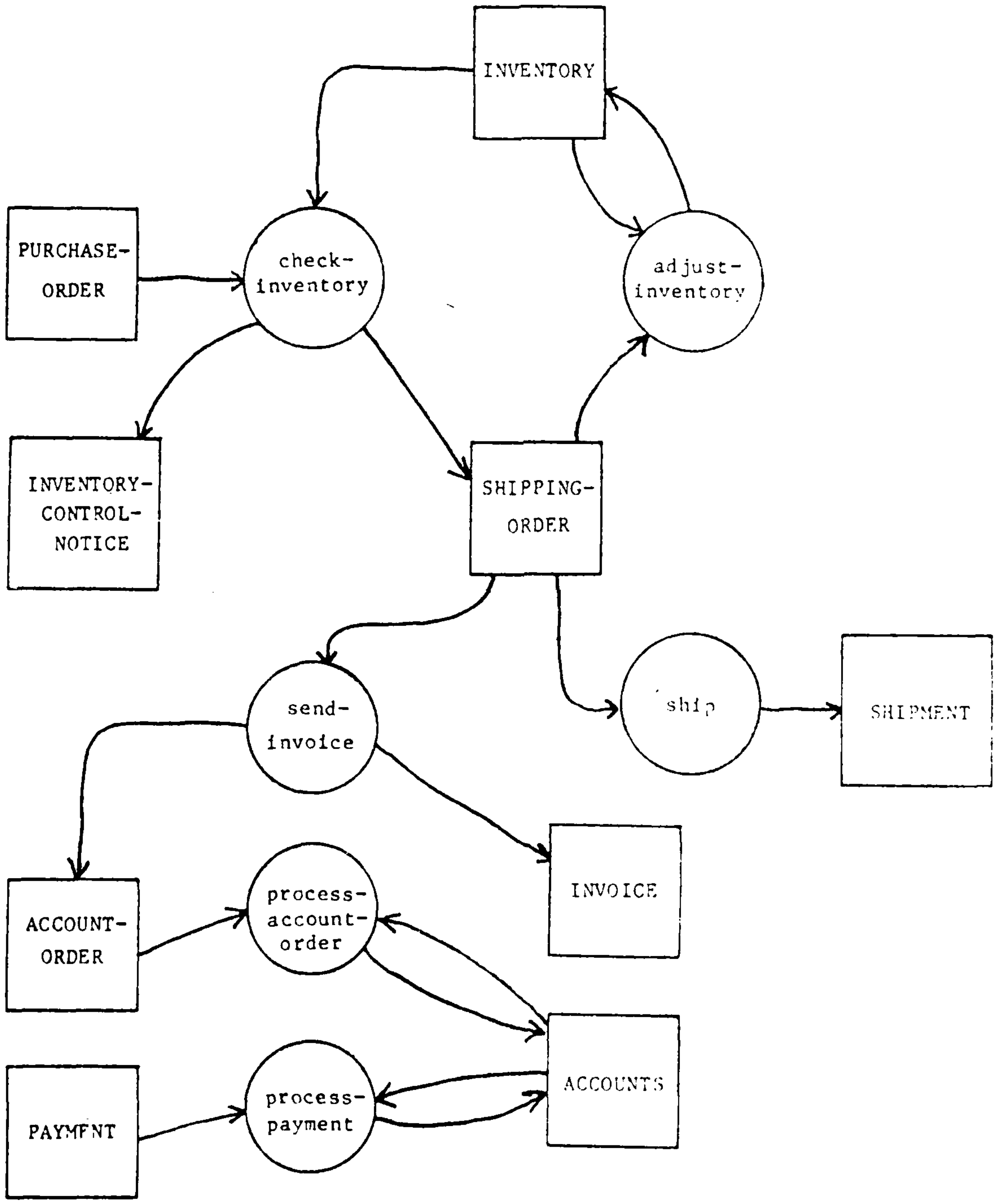


Figure 6. A dataflow diagram for filling orders from an inventory.

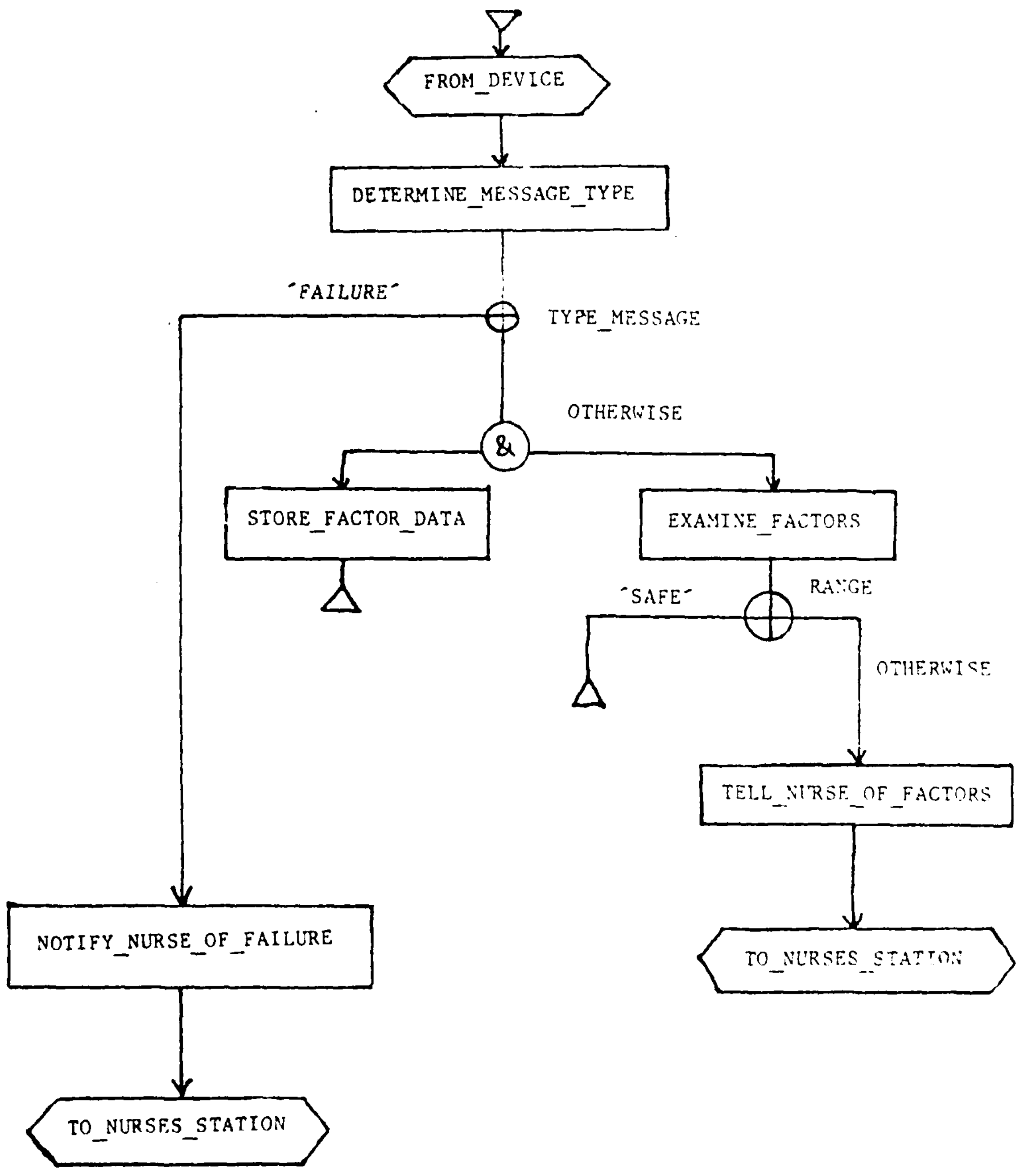


Figure 7. A stimulus-response path (part of a patient-monitoring system) specified in RSL (from [Alford 77]).

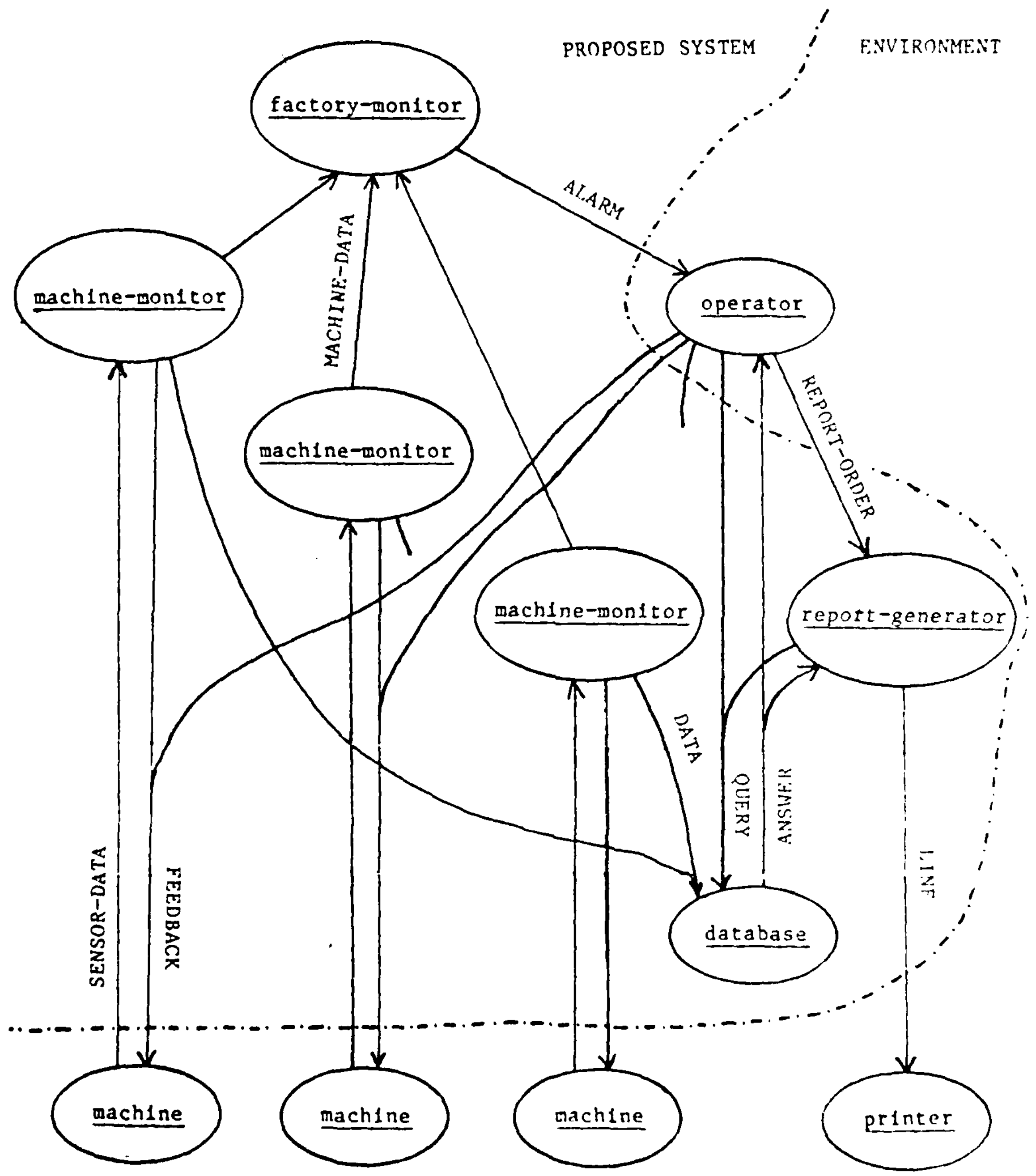


Figure 8. Process structure of the process-control system.

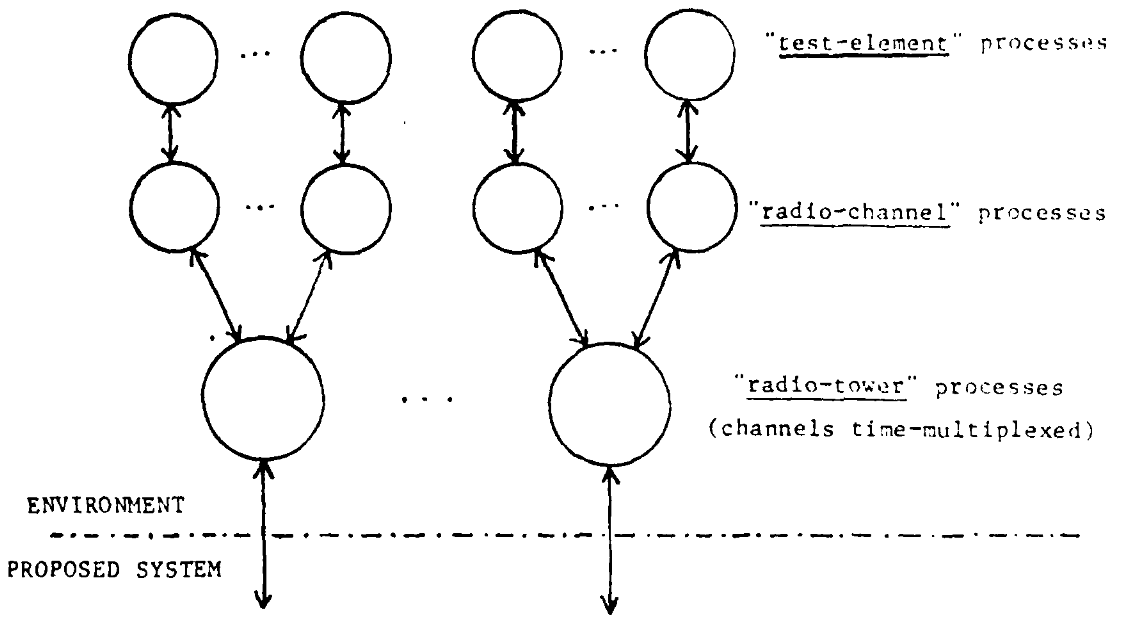


Figure 9. The environment of the AFWET system, including resource requirements.

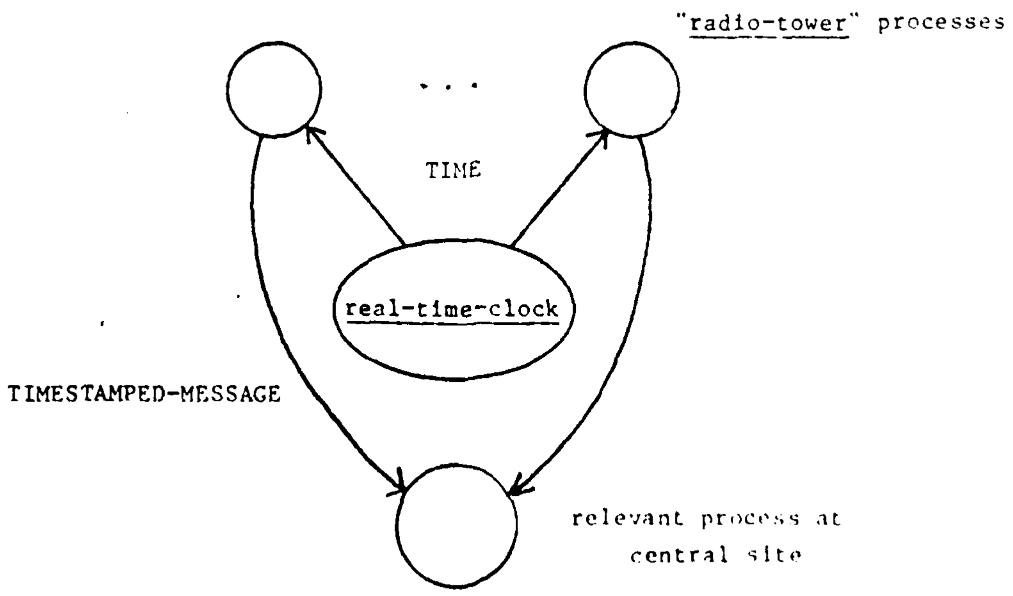


Figure 10. Specification of incoming message timestamps in AFWET.

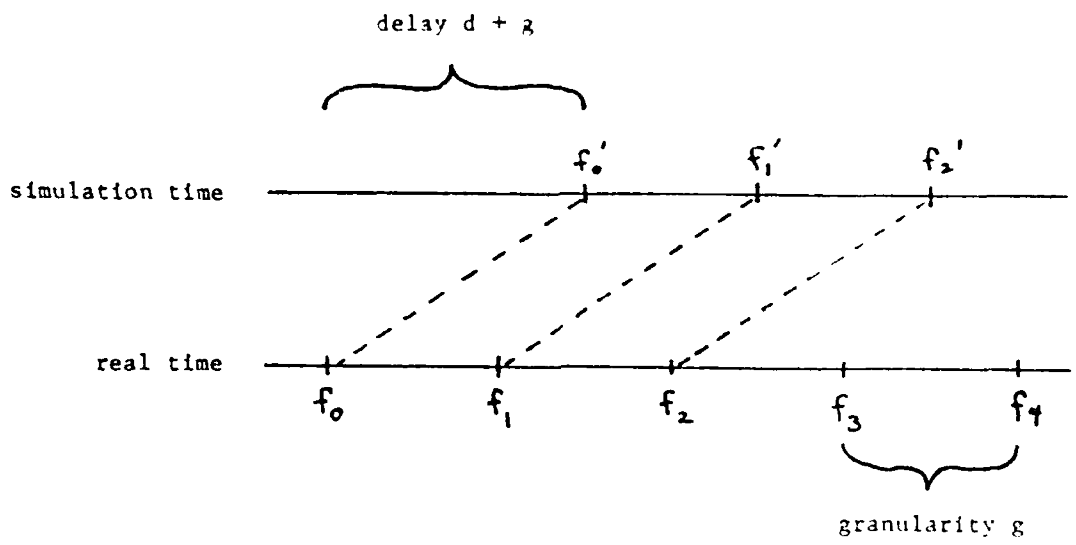


Figure 11. The time scheme of AFWET.

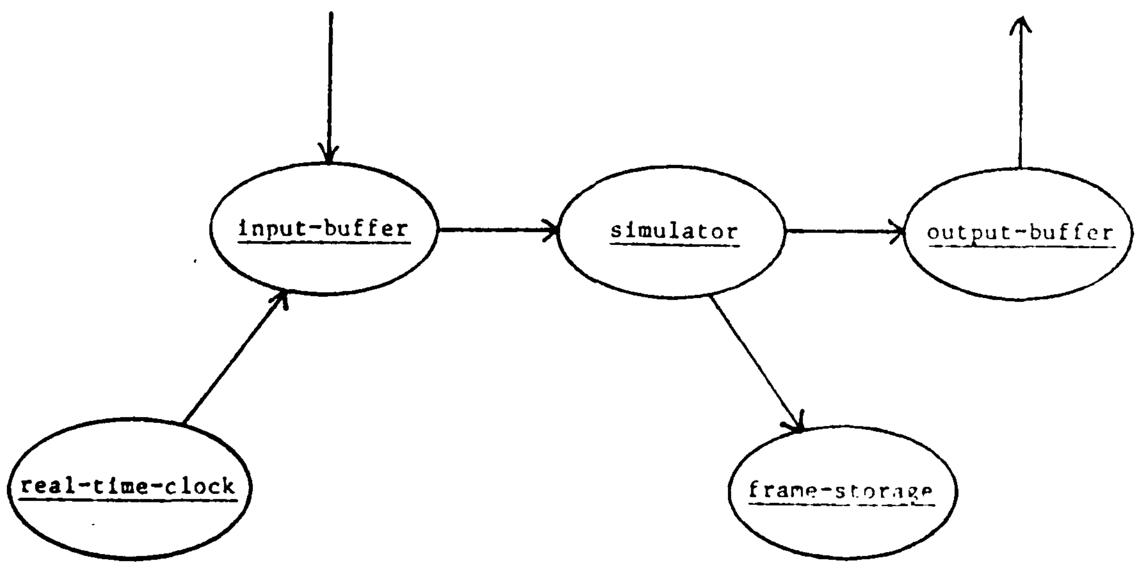


Figure 12. AFWET processes involved in simulation.

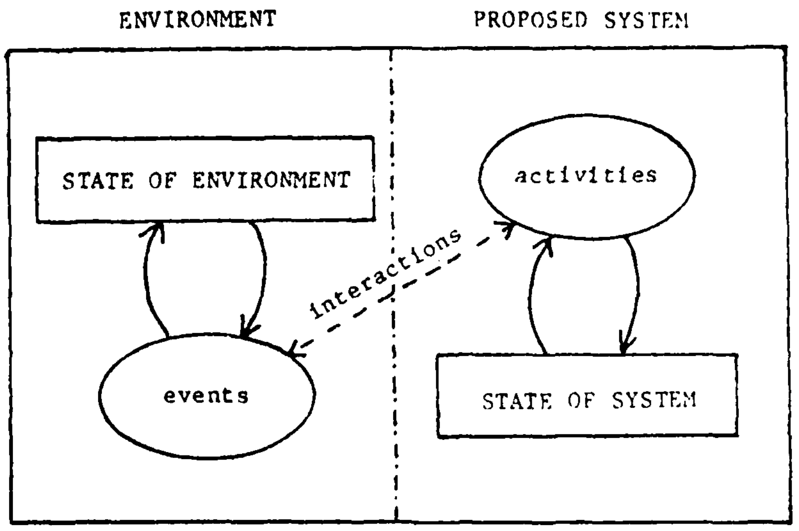
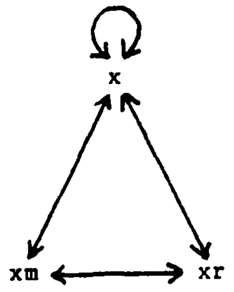


Figure 13. The conceptual model underlying both process-oriented and data-oriented approaches.



	synchronizing	free-running
self-matching	x	impossible
non-self-matching	xm	xr

(a) possible matches within a class

(b) derivation of the types

Figure 14. The three types of exchange functions.

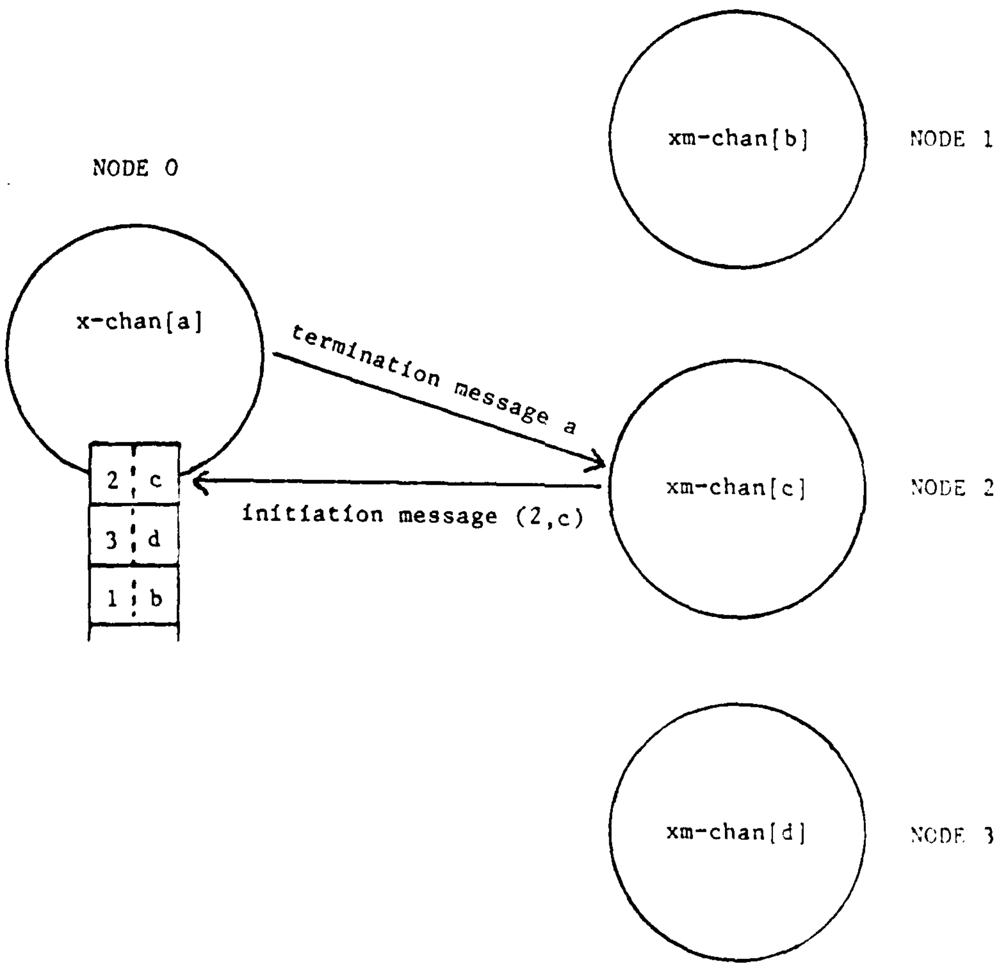


Figure 15. Distributed implementation of exchange matching.